SAIA: Un style architectural pour assurer l'indépendance vis-à-vis d'entrées / sorties soumises à des contraintes temporelles

Julien Deantoni

N° ordre : 2007-ISAL-0060

Institut National des Sciences Appliquées de Lyon

École Doctorale Informatique et Informations pour la Société

SAIA : un style architectural pour assurer l'indépendance vis-à-vis d'entrées / sorties soumises à des contraintes temporelles

THÈSE

présentée et soutenue publiquement le 12 octobre 2007

pour l'obtention du

Doctorat de l'Institut National des Sciences Appliquées de Lyon

(spécialité informatique)

par

Julien DeAntoni

Composition du jury

Rapporteurs :	Jean-Marc Jézéquel	Professeur à l'université de Rennes - IRISA
	Lionel Seinturier	Professeur à l'université de Lille 1 - LIFL
Examinateurs :	Jean-Philippe Babau	Maître de conférence à l'INSA-Lyon - CITI (directeur de thèse)
	Sébastien Gérard	Chercheur au CEA-Saclay - LIST
	Yvon Trinquet	Professeur à l'université de Nantes - IRCCyN
	Stéphane Ubéda	Professeur à l'INSA-Lyon - CITI
	François Vernadat	Professeur à l'INSA-Toulouse - LAAS

Laboratoire CITI : Centre d'Innovations en Télécommunications et Intégration de services

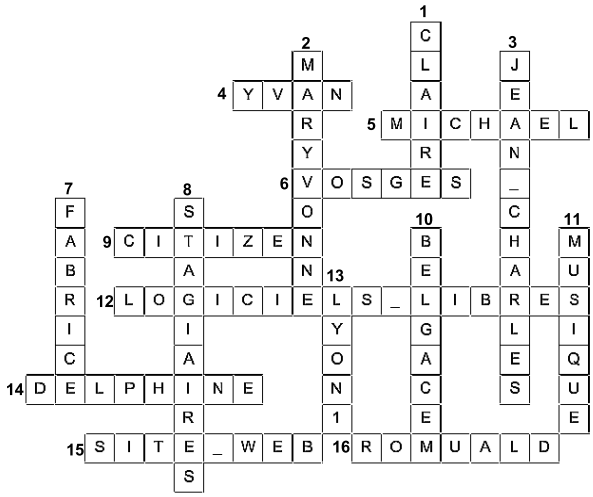
Remerciements

Une thèse, trois ans de travail et enfin la rédaction de ce manuscrit. Il est clair que cet aboutissement doit énormément au soutien des gens qui nous ont entourés et compris, dans le cadre professionnel aussi bien que dans le cadre personnel. Je tiens donc à les remercier au travers de cette page.

Je tiens d'abord à remercier Jean-Philippe Babau, directeur et encadrant de cette thèse, pour les trois années passées à ses côtés. Il a su m'orienter dans ce petit monde qu'est la recherche et faire preuve de patience dans les moments difficiles. Je le remercie pour ses conseils, sa compréhension et l'autonomie qu'il m'a laissé dans mon travail. Il est pour beaucoup dans la réussite de cette thèse.

Je tiens également à remercier chacun des membres du jury pour avoir accepté de jouer ce rôle et en particulier, Jean-Marc Jézéquel et Lionel Seinturier pour les conseils avisés qu'ils m'ont procurés en tant que rapporteurs de cette thèse.

Enfin, pour ne pas utiliser un style trop répétitif et afin d'apporter la touche d'originalité souvent recherchée dans les remerciements, la suite de mes remerciements est présentée, page suivante, sous forme de mots croisés.



Horizontal

4 Amis : aussi appelé monsieur dépoireaux, on a passé pas mal de temps à parler de tout et surtout de rien.

5 Amis/collègues : une rencontre originale pour quelqu'un d'original avec qui il fait bon discuter.

6 Autres : là où se mettre au vert prend tout son sens, c'est également un endroit peuplé d'amis.

9 Collègues : Menés par Stéphane Ubéda qui m'a accueilli dans le laboratoire CITI et à qui j'exprime ma gratitude, ils m'ont tous donné les moyens d'effectuer cette thèse dans les meilleurs conditions.

12 Autres : utilisés au quotidien, je pense qu'ils apportent beaucoup et en particulier au monde académique.

14 Famille : elle m'a soutenu depuis le début de mes études, au moins une heure par semaine au téléphone ;-) Merci ! ! !

15 Autres : wikipedia, wordreference, radioblogclub, scholargoogle en font parti.

16 Amis/collègues : aussi fort en info qu'aux jeux de cartes, son foie doit me haïr...

Vertical

1 Amies++ : son soutien journalier et ses relectures ne sont qu'une infime partie du bon temps passé à ses côtés.

2 Famille : sans elle je ne serais rien, au sens propre comme figuré ! Je lui dois énormément.

3 Amis/collègues : Pas facile d'être thésard après lui, malgré cela ce fut un bonheur de le connaître et de discuter avec lui.

7 Famille : que de bon temps passé avec lui à discuter et à bosser comme des vrais chefs ed' chantier.

8 Collègues : Sylvain, Pierrick, Pitou, Aurélien, Adrien, Samarth, ce fut un plaisir de les encadrer ou simplement de les accueillir au bureau.

10 Amis/collègues : après 4 ans de vie commune au bureau, je peux lui dire un grand *Chokran*.

11 Autres : Noir Désir, Léo Ferré, Les béruriers noirs, Tracy Chapmann, Alain Souchon ne sont que des exemples.

13 Autres : Université dans laquelle j'ai eu le bonheur d'être moniteur et ainsi, d'effectuer mes premières armes en tant qu'enseignant.

Merci à tous ! ! !

Table des matières

1

Introduction

Les ordinateurs personnels sont la manifestation la plus visible de l'arrivée massive des ordinateurs dans notre quotidien. Pour autant la quasi-totalité des processeurs que nous utilisons sont intégrés dans des objets de notre quotidien. Depuis les machines à laver, en passant par les climatisations ou encore par les moyens de locomotions, l'informatique prend une place de plus en plus enfouie dans notre vie de tous les jours. Ces systèmes informatiques appelés systèmes embarqués sont différents des ordinateurs personnels (PC, Mac) pour plusieurs raisons.

La première différence concerne leur utilisation. Un ordinateur personnel permet de réaliser plusieurs activités aussi diverses et variées que la programmation de logiciels, des calculs complexes, la lecture de fichiers multimédia ou encore la rédaction de documents de bureautique. Contrairement à cela, si un système tel qu'un four intègre un système informatique, celui-ci ne fournit que les fonctionnalités en rapport avec le four ; soit le contrôle de la cuisson des aliments. On parle alors de système dédié, c'est-à-dire un système dont les fonctionnalités logicielles sont intimement liées aux fonctionnalités de l'objet sur lequel elles s'exécutent.

Une autre différence entre les systèmes embarqués et les ordinateurs personnels est qu'ils ne disposent pas d'écrans classiques, de claviers ou de souris. La communication entre l'homme et de tels systèmes, lorsqu'elle est possible, est plus limitée qu'avec un ordinateur personnel. On parle alors de systèmes enfouis, c'est-à-dire invisible à l'utilisateur. De ce fait, la maintenance en est plus complexe car le système peut ne pas posséder de touche "redémarrer" ou "*reset*". Ces systèmes sont donc soumis à des vérifications plus poussées que les ordinateurs personnels afin d'éviter un arrêt inopiné, les rendant potentiellement dangereux ou inutilisables. Si dans le cas du four, la cuisson risque simplement d'être mauvaise, dans le cas du système de freinage d'une voiture, les conséquences peuvent être graves et mettre des vies en danger ; on parle alors de systèmes critiques.

Une autre différence notable est qu'un système embarqué communique avec un processus

physique dont il doit assurer le contrôle. Par exemple l'ABS (*Anti Blocage System*) d'une voiture ne communique pas directement avec l'utilisateur de la voiture mais communique de manière permanente avec l'environnement physique ; c'est-à-dire avec la roue et le frein de la voiture. On parle alors de système de contrôle de processus. Un système de contrôle de processus est en charge de réguler de manière fiable un processus de l'environnement physique. Pour cela il récupère les informations en provenance de l'environnement à l'aide de capteurs et il agit sur l'environnement physique à l'aide d'actionneurs.

Le fait qu'un système embarqué soit lié à un processus et donc à la dynamique de ce processus en fait un système temps réel ; c'est-à-dire un système dont la validité dépend non seulement de l'action entreprise mais aussi du temps dans lequel l'action est entreprise. À titre d'illustration, la commande de freinage d'un véhicule est dite correcte si elle ralentit le véhicule mais aussi si elle est effectuée dans un temps borné après l'appui sur la pédale de frein.

Dans le cadre de cette thèse, nous nous intéressons au développement des systèmes embarqués, dédiés au contrôle de processus. Plus précisément, nous nous intéressons à la partie logicielle de ces systèmes en assurant leur correction.

Les caractéristiques énoncées précédemment impliquent des différences entre le développement des systèmes de contrôle de processus temps réel et le développement d'un système classique sur deux points principaux. D'une part la criticité des systèmes et leurs contraintes temporelles imposent une vérification formelle. D'autre part, l'aspect dédié des choix technologiques (en terme de système d'exploitation, de moyens d'interaction avec l'environnement, de processeur, de capacité mémoire, de médium de communication, ...) est spécifique à chaque domaine, voire à chaque système. On peut noter qu'un des aspects spécifiques essentiel dans l'architecture matérielle des systèmes de contrôle de processus temps réel porte sur les moyens d'interaction de l'application avec son environnement via les capteurs et les actionneurs. Ces choix impactent lourdement le comportement temporel des systèmes et doivent donc faire l'objet d'une attention particulière. Afin de maîtriser finement les paramètres temporels lors du développement et aboutir à un système correct, les méthodes utilisées sont souvent centrées sur l'implémentation. Toutes ces spécificités font des systèmes de contrôle de processus temps réel des systèmes difficiles à réutiliser ou à faire évoluer. Pour autant, afin d'être réactif aux besoins du marché, ces systèmes doivent aujourd'hui pouvoir évoluer rapidement, que ce soit au niveau de la partie matérielle ou de la partie logicielle qui y est implantée. Devant la multiplication et la complexité croissante de ces systèmes, assurer une réutilisation sûre de certaines fonctionnalités logicielles de ces systèmes doit aider à améliorer le développement. En particulier, comme dans les systèmes classiques, un besoin de plus en plus exprimé par les industriels est de pouvoir développer une application logicielle en s'affranchissant de la cible matérielle et en particulier en s'affranchissant des capteurs et des actionneurs. Ceci est d'autant plus vrai que les premières phases du développement des applications logicielles ne sont pas effectuées sur une cible matérielle réelle mais sur la cible matérielle simulée. Permettre de s'affranchir de la cible matérielle lors du développement offre deux avantages certains :

- une application développée et validée sur une cible matérielle simulée peut être déployée sur une cible réelle sans modification ;
- une application peut être déployée sur des cibles matérielles différentes.

Cependant, puisque la qualité de l'interaction entre l'application et le processus impacte fortement la validité du système, il est nécessaire de pouvoir qualifier la qualité d'interaction nécessaire à une application pour être réutilisée correctement. Ces différentes préoccupations

amènent à se poser une série de questions :

- Comment développer une application de contrôle de processus indépendamment d'une technologie spécifique de capteurs et d'actionneurs ?
- Comment spécifier et valider la qualité d'interaction nécessaire à l'application pour assurer un contrôle correct ?
- Et enfin, comment s'assurer qu'une technologie de capteurs et d'actionneurs réels respecte la qualité d'interaction spécifiée par l'application ?

Considérer ces questions influe sur le processus de développement. Le développement d'un système passe par un enchaînement d'étapes ayant chacune un objectif différent dans la conception du système. Définir ces différentes étapes, leurs enchaînements et objectifs revient à gérer le développement d'un système informatique. On regroupe les études liées à ce domaine à un vaste domaine visant à améliorer la qualité logiciel et nommé génie logiciel. De nombreux principes de génie logiciel ont été proposés dans la littérature pour améliorer le développement de systèmes complexes et en particulier pour aider la réutilisation correcte des applications. Parmi ces travaux on retrouve les études autour de la structuration de logiciel (*software architecture*) et les approches dirigées par les modèles. La structuration de logiciel permet de raisonner sur une vue de haut niveau d'un système logiciel et ainsi offre un support pour la prise en considération de l'évolutivité et de la réutilisation des systèmes logiciels. Les approches dirigées par les modèles, quand à elles, permettent de définir les concepts d'un domaine à partir d'un ou plusieurs modèles placés au centre de l'activité de développement. Ces approches ont déjà montré leurs bénéfices pour la réutilisation, lors du développement, de systèmes non dédiés et non temps réel. Certains travaux actuels concernent l'application de ces principes dans le cadre des systèmes dédiés temps réels. Cependant, dans les systèmes visés par notre étude, les moyens spécifiques d'interaction avec l'environnement sont rarement traités.

Le but de cette thèse est de traiter les questions énoncées précédemment en utilisant les principes de génie logiciel liés à la structuration de logiciel et aux approches dirigées par les modèles. Plus particulièrement, l'étude s'attache à fournir, d'une part des principes de structuration permettant de développer une application de contrôle de processus indépendamment d'une technologie de capteurs et d'actionneurs ; et d'autre part à assurer, par une analyse formelle, la correction de l'application lors de son utilisation avec une technologie de capteurs et d'actionneurs spécifiques. Ces deux préoccupations ont été traitées en fournissant :

- un style architectural définissant des règles de structuration du logiciel via différents types de composants et des contraintes sur leur association permettant d'obtenir une structure analysable ;
- une formalisation des contraintes temporelles concernant : la communication d'une application de contrôle avec le monde extérieur et la communication de l'application de contrôle avec une plateforme spécifique de communication avec le monde extérieur ;
- une formalisation de l'établissement d'un contrat de qualité de service (QoS : *Quality of Service*) lors du déploiement de l'application de contrôle sur une plateforme de communication avec le monde extérieur,
- une proposition de méthode et d'outil pour faciliter la mise en œuvre du style architectural et des analyses formelles.

Cette thèse s'articule autour de cinq chapitres. Le deuxième chapitre commence par une présentation du contexte d'étude dans lequel s'inscrit cette thèse. Il permet de présenter les spécificités des systèmes visés ainsi que de définir les termes employés tout au long de cette thèse. Une fois le contexte présenté, un état de l'art composé de deux parties est proposé. La première partie de cet état de l'art propose un panel des concepts actuellement proposés pour

la modélisation d'une architecture logicielle, en particulier dans le domaine de l'embarqué. Cette partie est issue d'une synthèse des différentes approches existantes aussi bien dans le domaine du génie logiciel à base de composants (CBSE : *Component Based Software Engineering*) que dans le domaine des architectures logicielles (*Software Architecture*) proprement dit. La troisième partie de ce chapitre présente les structurations classiquement adoptées en informatique pour permettre la réutilisation correcte d'une même application sur plusieurs cibles.

Après une analyse critique de l'état de l'art, le quatrième chapitre décrit la proposition effectuée dans cette thèse, appelée SAIA pour *Sensors / Actuators Independent Architecture*. Afin de faciliter la lecture, ce chapitre commence par une synthèse permettant d'introduire les concepts de SAIA. Ces concepts sont décrits au travers des principes de structuration ainsi que des principes de gestion de la QoS. Ensuite, le modèle à composant utilisé est présenté avant de décrire en détail les différents types de composants définis ainsi que les services qui leurs sont associés. Une fois les types de composants présentés, une vision détaillée de la formalisation et de la gestion des contraintes temporelles pour l'établissement d'un contrat de QoS est fournie. Enfin, une conclusion rappelle les points clefs de l'approche.

Le cinquième chapitre propose ensuite une évaluation de l'approche proposée. Pour cela il commence par présenter l'outil de modélisation permettant de construire des applications conformes à SAIA. Ensuite, deux méthodes pour l'utilisation de SAIA sont présentées et illustrées au travers d'un exemple. La première méthode définit des processus incrémentaux de modélisation alors que la deuxième est focalisée sur la mise au point des paramètres temporels système. Pour finir ce chapitre, la mise en œuvre de SAIA est présentée au travers des modèles et des implémentations réalisés dans le cadre de deux concours internationaux de programmation temps réel. Le premier concours a permis de s'assurer de la faisabilité de l'implémentation de l'architecture modélisée. Le deuxième concours a permis d'évaluer la propension de réutilisation des modèles SAIA en réutilisant les modèles du premier concours sur une plateforme de communication différente.

Enfin, le sixième et dernier chapitre donne les conclusions, ainsi que des perspectives pour les travaux menés durant cette thèse.

2

État de l'art

Sommaire

2.1 Domaine d'étude

2.1.1 Introduction

L'étude menée dans cette thèse s'intéresse aux systèmes de contrôle de processus. Un système de contrôle de processus est un système informatique dont le fonctionnement est assujetti à l'évolution dynamique de l'état de l'environnement qui lui est connecté et dont il doit contrôler le comportement.

Dans de tels systèmes, on distingue quatre entités principales (cf. figure 2.1) :

le système de contrôle : c'est l'entité qui comprend l'application de contrôle ainsi que le support lui permettant de s'exécuter. C'est l'association support et logiciel qui permet d'obtenir les informations sur le processus via des capteurs et d'agir sur celui-ci à l'aide d'actionneurs.

le processus : c'est l'entité à contrôler. Son état peut être mesuré par un ou plusieurs capteurs de même qu'il peut être modifié par un ou plusieurs actionneurs. Le processus évolue au cœur de son environnement.

l'environnement physique du processus : c'est l'ensemble des phénomènes physiques qui n'appartiennent pas au processus mais dont l'évolution impacte le comportement du processus.

l'opérateur : c'est une personne ou un système extérieur fournissant des directives que le système doit réaliser sur le processus. Il peut être informé sur l'état du processus ainsi que sur l'état du contrôle.

Dans la suite du document, l'ensemble environnement physique, processus et opérateur est désigné par l'expression "monde extérieur".

On distingue plusieurs types d'informations échangées entre le monde extérieur et le système. Les informations dites "en entrée" sont consommées par le système et celles dites "en sortie" sont produites par le système. En entrée, on distingue les *Mesures* effectuées sur le processus ou sur l'environnement, et les *Consignes* fournies par l'opérateur.

Une *Mesure* reflète l'état du processus ou de l'environnement. Une *mesure* est un flot infini d'occurrences notées f représentatif d'une grandeur physique appartenant au monde extérieur et notée φ. Chaque occurrence du flot est notée f_i et correspond à une occurrence de φ notée φ_i où i représente le numéro de l'occurrence entre 1 et l'∞. Elles sont classiquement déclinées en données et en événements. L'occurrence d'une donnée est caractérisée par une date d'acquisition notée $@f_i$ et une valeur notée $V(f_i)$ alors que l'occurrence d'un évènement est caractérisée seulement par sa date d'arrivée dans le système. On parle alors de flot de données ou de flot d'événements.

Toujours en entrée, une *Consigne* est un flot de données ou d'événements servant de directive ou d'objectif pour le contrôle de processus. Dans notre étude, toute information en entrée est acquise par l'intermédiaire d'entités appelées capteurs (cf. figure 2.1).

Les informations circulant en sortie du système sont appelées *Commandes*. Les *Commandes* sont un flot de données ou d'événements. Ces flots d'informations servent à commander les actionneurs (cf. figure 2.1) qui transforment chaque occurrence en une action physique et modifient

ainsi l'état du processus.

À titre d'exemple, prenons le cas du contrôle d'une voiture. Le système est représenté par : les capteurs fournissant les *Mesures* (vitesse, température moteur, etc.), les capteurs fournissant les *Consignes* (position de la pédale d'accélération, de freins, etc.), un système de contrôle qui calcule les *Commandes* et les actionneurs qui réalisent les *Commandes* (niveau d'admission électronique, pression de la mâchoire de frein, puissance d'un moteur de ventilateur, etc.).

Dans ce cas, le processus est la voiture en elle-même, c'est-à-dire sa carrosserie, ses roulements, ses roues, la boîte de vitesse, etc. L'environnement de la voiture est constitué de la route, sa signalétique, des autres voitures, etc. L'opérateur est ici le conducteur (réel mais pourquoi pas automatique) qui actionne les pédales de freins, d'accélération et autres afin de fournir les *Consignes* au système.

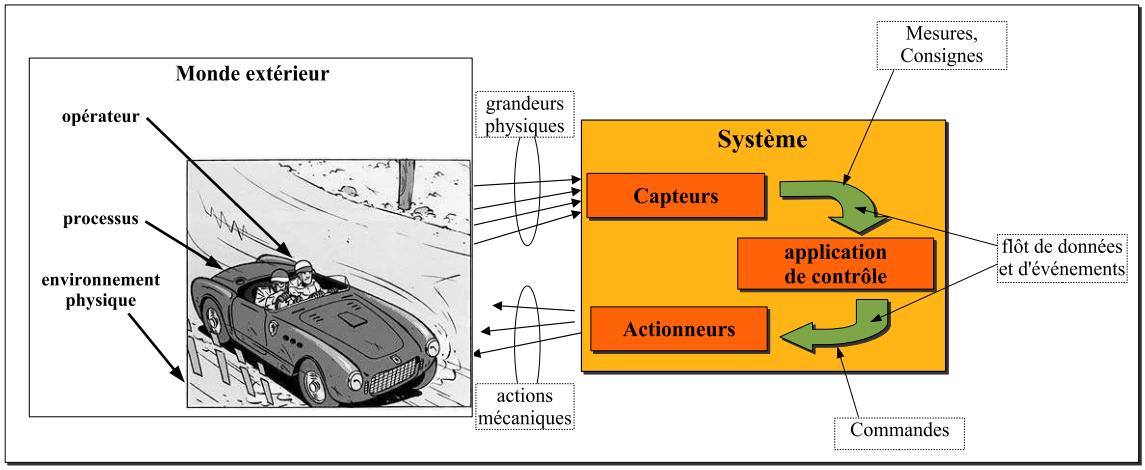


FIG. 2.1 – Représentation d'un système de contrôle de processus

Il est à noter que pour certaines *Consignes*, lorsqu'elles sont dictées par un utilisateur, le capteur correspondant peut faire partie d'une interface homme machine (IHM). Les approches réalisées autour des IHMs sont nombreuses et font l'objet d'études particulières. Dans ce document les informations en provenance ou à destination d'IHMs sont considérées comme des informations en provenance de capteurs ou à destination d'actionneurs.

2.1.2 Aspects temporels des systèmes de contrôle de processus

Les systèmes de contrôle de processus sont des systèmes temps réel dans le sens où leur dynamique est dictée par la dynamique du processus qu'ils contrôlent. Afin d'assurer un contrôle correct du processus, la qualité des informations échangées entre l'application de contrôle et le monde extérieur est essentielle. Ces informations sont donc sujettes à une description de leur qualité ; description essentiellement temporelle. [105, 66] mettent l'accent sur deux propriétés temporelles importantes : La loi d'arrivée et la loi de retard. Lorsqu'une fusion d'informations intervient en un point, il est également important de connaître la loi de corrélation temporelle entre les informations fusionnées. La définition informelle de ces propriétés temporelles est donnée dans la suite de ce chapitre.

Sur les *Mesures*, les *Commandes* et les *Consignes*, la loi d'arrivée est la loi exprimant le temps écoulé entre deux occurrences successives d'une *Mesure*, d'une *Commande* ou d'une *Consigne* dans son flot.

Pour les *Mesures* et les *Consignes*, la loi de retard exprime la loi sur les temps écoulés entre la date à laquelle une information est présente dans l'environnement et la date à laquelle l'occurrence correspondante est considérée dans le système.

Pour les *Commandes*, la loi de retard est la loi représentant le temps écoulé entre la date où le système produit la *Commande* et la date à laquelle la *Commande* ou l'action physique correspondante est considérée.

Pour toutes les informations dans le système, la loi de corrélation temporelle est la différence d'âge entre les informations fusionnées, au moment de la fusion.

D'autres propriétés peuvent aussi être considérées pour ces systèmes, telles que la précision sur les données ou encore les échéances de bout en bout, qui spécifie le temps acceptables entre l'acquisition d'une information en entrée et la réalisation de l'action physique correspondante en sortie du système. Dans notre étude, nous nous concentrons sur la loi d'arrivée, la loi de retard et la loi de corrélation temporelle. Ce choix s'appuie sur le constat de [129, 105, 66, 104, 77], qui montre que si les lois temporelles caractérisant les flots d'informations en provenance ou à destination des capteurs et des actionneurs ne sont pas prises en compte lors du développement, elles peuvent rendre le système à l'exécution instable. Plus précisément, les lois temporelles influent sur la qualité de contrôle du système. C'est-à-dire sur la précision du contrôle sur le processus par rapport à une *Consigne* donnée.

Nous venons de présenter les spécificités des systèmes de contrôle de processus. Ces spécificités influent sur le processus de développement dont la démarche classique ainsi que les tendances actuelles sont présentées dans la section suivante.

2.1.3 Développement des systèmes de contrôle de processus

Lors du développement d'un système de contrôle de processus, il est primordial de s'assurer que la qualité de contrôle du système est respectée. Nous avons vu que cela revient, entre autres, à maîtriser les contraintes temporelles du système. Pour ce faire, les approches dédiées au développement des systèmes de contrôle de processus sont différentes des approches de développement des systèmes informatiques classiques tels que les systèmes d'informations. En effet, afin de permettre une maîtrise fine des propriétés temporelles, les parties logicielle(s), matérielle(s) et physique(s) des systèmes de contrôle de processus sont fortement couplées dès les premières étapes du développement.

Classiquement, les approches de développement commencent par la modélisation du processus physique à contrôler ainsi que des différents capteurs et actionneurs permettant la communication avec le processus. Cette modélisation est basée sur des outils tels que Matlab/Simulink [80, 82], ASCET-SE [59] ou encore Dspace [41], permettant la modélisation de systèmes physiques. Il est alors possible de développer l'application de contrôle d'après les spécifications du système. Ce

développement passe essentiellement par l'utilisation de modèles ayant des fondements mathématiques forts tels que les modèles réalisables dans Simulink/Stateflow. Ces modèles peuvent alors être mis au point et validés vis-à-vis de leurs fonctionnalités et de la qualité de contrôle recherchées. Cette étape est réalisée par une phase de simulation incluant le modèle physique du processus à contrôler, le modèle des capteurs et des actionneurs ainsi que le modèle de l'application de contrôle. Une fois le modèle de l'application de contrôle mis au point, la dernière étape du développement consiste à implanter ce modèle ; c'est-à-dire à créer le code binaire permettant de réaliser les fonctionnalités décrites dans le modèle de l'application de contrôle. Depuis une dizaine d'années, des logiciels tels que SCADE [39] ou encore Matlab *real time workshop* [81] permettent de générer automatiquement le code. De plus, ces logiciels permettent de spécifier le modèle de tâches sous-jacent et de vérifier sa conformité par rapport aux contraintes temporelles.

Ces étapes découlent alors sur un système développé et mis au point pour un contexte matériel et physique donné. Il est alors très difficile de réutiliser la partie logicielle de tels systèmes dans un contexte matériel différent [5].

Partant de ce constat, des initiatives de standardisation commencent à apparaître. Par exemple, dans le domaine de l'automobile, on peut citer le standard Autosar [7] qui, par l'ajout d'une couche logicielle, permet d'homogénéiser l'accès au matériel bas niveau tels que les unités de calcul ou le réseau. Très récemment, la célèbre multinationale américaine de solutions informatiques Microsoft s'attaque au manque de flexibilité dans le développement des applications robotiques au travers de l'initiative Microsoft Robotic Studio [87]. Leur but est alors de créer une chaîne complète de développement facilitant le développement de systèmes robotiques.

2.1.4 Conclusion du domaine d'étude

Les modèles actuellement utilisés pour la modélisation d'un système de contrôle permettent de s'assurer de la correction du système. Cependant, afin de permettre la réutilisation de certaines parties logicielles, il est nécessaire de raisonner sur la structure de ces modèles. La structuration ne doit pas remettre pas en question l'utilisation de modèles tels que ceux manipulés dans Matlab/Simulink ou SCADE ; elle doit apporter de la qualité au sens du génie logiciel (réutilisation, capacité d'analyses, etc). Elle ne doit également pas remettre en question la validation de ces systèmes ainsi que la génération de leur code.

Ainsi, nous nous intéressons dans un premier temps aux techniques de structuration des logiciels découlant de certains principes de génie logiciel. Une fois les concepts de ces techniques abordés, nous détaillons les approches permettant de découpler les fonctionnalités d'un logiciel des services d'accès au matériel sur lequel elles s'exécutent.

2.2 Architecture et génie logiciel

Le concept de génie logiciel a émergé il y a maintenant un peu plus de trois décennies, lorsque le Comité Scientifique de l'OTAN a organisé successivement deux conférences tenues respectivement à Garmisch-Partenkirchen en octobre 1968 et à Rome en octobre 1969 [19, 91].

Les textes des exposés et des comptes rendus de discussions figurant dans les actes de ces deux conférences sont souvent référencés pour leur étonnante actualité. Les grands sujets débattus à l'heure actuelle y sont tous présents, notamment les composants et la réutilisation du logiciel, la notion de cycle de vie, la mise en place de processus de validation et de vérification, les techniques formelles, la conduite de projet ou l'estimation des coûts et des délais.

Cette thèse s'interroge sur la manière de développer les systèmes de contrôle de processus afin de faciliter la réutilisation d'une même application de contrôle sur différentes plateformes en termes de capteurs et d'actionneurs. Cette séparation doit permettre de garantir la correction d'une application de contrôle avant son déploiement sur un support réel. Parmi toutes les problématiques adressées par le génie logiciel, la structuration de logiciel et les analyses que cela autorise semble essentielle à l'atteinte de ces objectifs. Cette problématique, toujours d'actualités, a été étudiée dès la fin des années 60 par Dijkstra. Celui-ci expose certains principes de structuration [38] comme l'organisation en couche hiérarchique et l'abstraction permettant de faciliter le développement et la validation de systèmes informatiques. En particulier, il souligne que cette organisation particulière permet de réaliser des tests sans explosion des états. Toutefois, ce n'est que dix ans plus tard que la complexité des systèmes logiciels a fait émerger l'idée de conception architecturale [36]. Cette dernière idée distingue la notion de conception architecturale (*Programming-in-the-large*) de la notion de conception détaillée (*Programming-in-the-small*). Enfin, c'est dans les années 90 que la notion d'architecture logicielle en tant que "perspective haut niveau d'un système logiciel" devient une discipline à part entière [56].

2.2.1 Définitions

Il existe à l'heure actuelle de nombreuses définitions se rapportant au concept d'architecture logicielle (*software architecture*). Chacune des définitions parmi la centaine recensée [1], est colorée par le domaine duquel elle découle. Nous retenons, dans le cadre de cette thèse, deux définitions nous apparaissant suffisamment générales et pertinentes.

Dans le standard 1471 [55] datant de 2001, l'IEEE définit une architecture logicielle comme :

définition 1 "*The fundamental organization of a system embodied in its components, their relationships to each other, and the environment, and the principles guiding its design and evolution*"

Une architecture logicielle est donc définie par l'IEEE comme l'organisation principale d'un système à l'aide de composants, de la manière de connecter les composants entre eux ainsi que par les principes qui en guident la conception et l'évolution.

De son côté, le SEI (*Software Engineering Institute*) propose dans [52] une définition qui se veut être une synthèse de nombreuses définitions antérieures :

définition 2 "*[...] While there are numerous definition of software architecture, at the core of them is the notion that the architecture of a system describes its gross structure. This*

[1] http ://www.sei.cmu.edu/architecture/definitions.html

> *structure illuminates the top level design decisions, including things such as how the system is composed of interacting parts, where are the main pathways of interaction, and what are the key properties of the parts. Additionally, an architectural description includes sufficient information to allow high level analysis and critical appraisal*"

Cette définition considère une architecture comme la description de la structure d'un système. Cette structure est le reflet des décisions prises lors de la modélisation de haut niveau ; c'est-à-dire qu'elle précise quelles sont les différentes parties d'un système, quelles sont les interactions entre ces parties ainsi que les propriétés essentielles de chacune de ces parties. De plus, cette définition met l'accent sur le fait qu'une architecture doit pouvoir permettre d'effectuer des analyses sur le système, ainsi qu'une évaluation des points critiques.

De manière plus générale, toutes les approches considèrent une architecture comme une structure "gros grains" d'un système. Cette structure permet de définir, à l'aide d'un assemblage de composants, les décisions relatives à la conception d'un système. Elle peut être raffinée et analysée afin de guider la mise en œuvre du système. Ainsi, une architecture logicielle peut être considérée comme un point pivot entre les exigences des spécifications, la conception et la mise en œuvre.

Puisque la description architecturale conditionne les choix relatifs à la conception d'un système, il est essentiel que celle ci soit validée pour assurer la cohérence des choix par rapport aux objectifs temporels, économiques, ou autres du système.

2.2.2 Bénéfices attendus

De nombreux bénéfices sont attendus du fait de la maîtrise des architectures logicielles. Puisqu'une architecture reflète la structure "gros grains" d'un système, elle permet à l'architecte de se concentrer en premier lieu sur l'organisation du système en différents blocs interagissant. Cette préoccupation efface les soucis techniques inhérents à chacun des blocs et fait ainsi apparaître une décomposition naturelle du problème en sous problèmes. On assiste ainsi à une séparation des préoccupations importante pour la maîtrise de la complexité [9, 52].

Cette séparation des préoccupations fournit une première conception du système où certains détails sont masqués. Elle permet ensuite de se concentrer sur les détails d'un composant sans avoir à se soucier des autres composants. Cela permet de faciliter la communication entre les différents intervenants d'un système en leur fournissant un niveau de détails adapté à leur tâche [36, 56, 9, 52].

De plus, la possibilité de contraindre l'organisation de la structure et de ses composants permet une mise en œuvre de principes de génie logiciel [58]. L'ensemble de ces contraintes, appelé un style architectural, permet d'obtenir certaines propriétés de qualité au sens du génie logiciel, comme la réutilisation, la capacité d'analyse, l'interopérabilité, etc. [89].

Enfin, associée à des techniques formelles de calculs, l'utilisation d'une architecture logicielle permet d'effectuer des analyses sur les diverses parties du système et sur leur composition [53, 52, 58]. Ces analyses "au plus tôt" de certaines propriétés doivent permettre les premières validations

du système. En détectant le non-respect de certaines propriétés, il est plus facile de revenir sur les choix architecturaux défaillants. Ces analyses doivent être faites à plusieurs stades du raffinement de l'architecture [6, 22].

On peut résumer les principaux bénéfices attendus lors de l'utilisation d'une architecture logicielle suivant quatre points. L'utilisation d'une architecture doit fournir :

- un cadre pour **maîtriser la complexité** d'un système (niveau d'abstraction puis raffinement) ;
- un support pour la **prise en compte de principes de génie logiciel** (utilisation de styles architecturaux) ;
- une base de **raisonnement à des fins d'analyse et de validation** (à tous les niveaux de raffinement et d'abstraction) ;
- une **documentation** du système facilitant sa mise en œuvre ainsi que sa communication.

Maintenant que nous avons vu les bénéfices attendus d'un raisonnement au niveau architectural, nous donnons les éléments nécessaires à l'établissement d'une architecture.

2.2.3 Décrire une architecture

Aux vues des études existantes et en se basant sur la synthèse effectuée dans [84], une architecture est caractérisée par les éléments suivants (cf. figure 2.2) :

les composants qui sont les briques de base d'une architecture ;

les interfaces qui sont associées à un composant et permettent d'accéder aux propriétés ou comportement(s) de celui-ci ;

les connecteurs qui permettent de spécifier les communications entre les interfaces ;

la configuration qui est un assemblage de composants, reliés au travers d'interfaces par des connecteurs ;

le style architectural qui est un ensemble de contraintes sur les configurations possibles ou acceptables ;

l'ADL *Architecture Description Language* qui est un langage permettant la description des éléments précédents.

Dans la suite de ce chapitre, nous présentons plus précisément chacun des concepts nécessaires à l'établissement d'une architecture logicielle, en particulier dans le domaine de l'embarqué et du temps réel. Les concepts sont organisés selon la liste précédente et s'appuient sur un état de l'art du domaine. Nous discutons ensuite des possibilités d'analyse et de mise en œuvre des architectures logicielles avant de conclure.

2.2.3.1 Les composants

Un composant est une unité de traitement ou de stockage de données. Il représente, avec les connecteurs, la brique de base utilisée pour décrire un système. À un composant est associé un comportement composé d'un ensemble de services (au sens de fonctions ou de méthodes). On distingue dans la majorité des approches deux manières de décrire le comportement d'un composant :

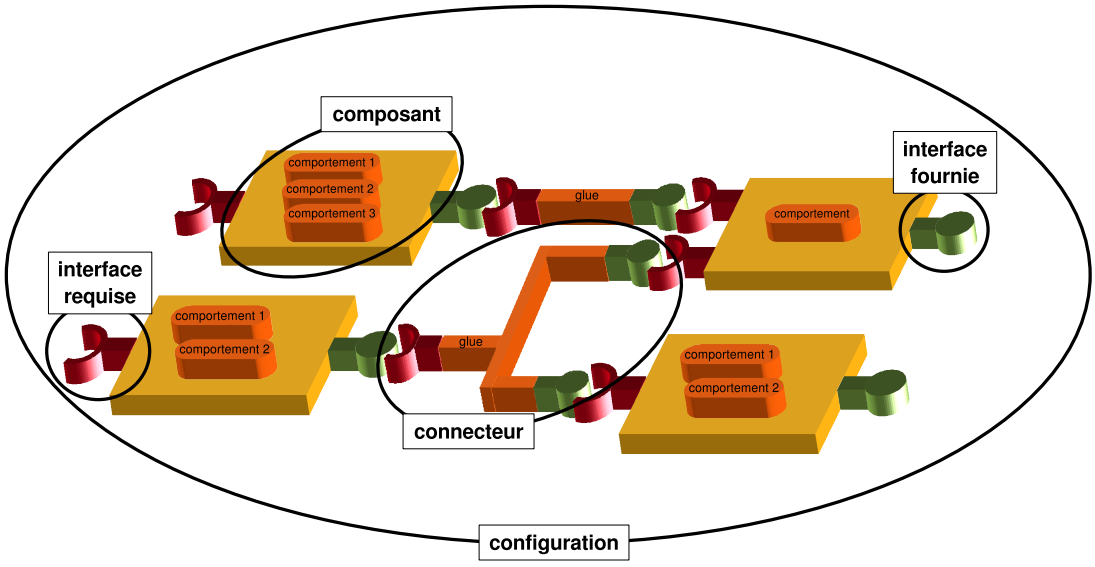


FIG. 2.2 – Exemple de représentation graphique d'une architecture logicielle

- les composants dit primaires possèdent un comportement décrit dans un langage spécifique (langage de programmation, langage formel, langage machine) ;
- les composants composites, lorsqu'ils sont pris en charge par l'approche, possèdent un comportement décrit à l'aide d'une configuration (c'est-à-dire d'un ensemble de composants interconnectés).

L'utilisation de composants composites permet de créer une hiérarchie de composants. Un composant composite est donc d'un niveau d'abstraction supérieur aux composants qui le constituent.

Le comportement d'un composant primaire peut être spécifié de différentes manières. La littérature met en avant trois catégories non exclusives :

1. Spécification du comportement par un (ou plusieurs) exécutable(s) binaire(s) ou sous forme de byte code (osgi [98], .Net [86], EJB [88], EAST-EEA [101], PACC [128]). Dans ce cas on parle de composants en boîte noire car le comportement peut être utilisé mais son code source, puisque (pré)compilé, reste caché. Le composant se doit alors d'inclure chacun des codes binaires correspondant aux cibles pour lequel il a été prévu.
2. Spécification du comportement à l'aide d'un (ou plusieurs) code(s) source(s) exprimé(s) dans un langage de programmation "classique" (koala [124], PBO [113], rubus [63], PECOS [57], Think [44], MetaH [125], EAST-EEA [101], VEST [112], giotto [61], AADL [99]). Dans le contexte des systèmes embarqués, le langage C est un choix récurrent pour la description du comportement. Dans ce cas on parle de composants en boîte blanche car le code source du composant est fourni et visible. Une phase de compilation est alors nécessaire pour permettre l'exécution d'un composant sur une cible donnée.
3. Spécification du comportement à l'aide d'un langage formel. Cette approche permet de réaliser diverses analyses sur le composant ou sur un assemblage de composants. On trouve dans la littérature les approches basées sur le langage CSP (Wright [2]) ou sur les automates à états finis (CLARA [40], MetaH [125], AADL [99], PECOS [57], COTRE [42],

C2 [37], PACC [128]). Toujours à des fins d'analyses, le comportement d'un composant peut être abstrait selon divers points de vue. Typiquement, d'un point de vue temporel, un comportement peut être abstrait par son pire (ou son meilleur) temps d'exécution afin d'effectuer des analyses d'ordonnançabilité (CLARA [40], MetaH [125], EAST-EEA [101], VEST [112], giotto [61]). Cette approche de description du comportement fournit un composant en boîte grise ; c'est-à-dire un composant ne fournissant pas le code source mais les propriétés nécessaires à son utilisation correcte ou à l'analyse de son comportement vis-à-vis d'une propriété donnée.

Puisque les trois catégories précédentes ne sont pas exclusives, beaucoup d'approches associent différentes descriptions possibles d'un comportement. En particulier, pour réaliser des analyses de conception, le comportement est décrit à l'aide d'un code binaire et/ou d'un code source et est caractérisé par des propriétés abstraites nécessaires à l'analyse (MetaH [125], EAST-EEA [101], VEST [112], giotto [61], PACC [128], koala [124], PECOS [57]) ou nécessaire au déploiement (osgi [98], .Net [86]).

Conclusion sur les composants Chacune des manières de décrire le comportement d'un composant possède des intérêts différents. La solution à retenir dépend donc des besoins exprimés par le domaine d'application, soit classiquement : l'exécution, le déploiement ou l'analyse.

Exécution :

Pour assurer l'installation et/ou le remplacement de composants lors de l'exécution, l'utilisation de code binaire permet d'éviter la phase de compilation. Par contre, son déploiement est limité aux cibles sur lesquelles le code binaire est exécutable. Il faut toutefois remarquer que certains codes sont exécutables sur plusieurs cibles (le byte code Java [115] par exemple). Il faut également remarquer que la description d'un comportement sous forme de code binaire évite de dévoiler le code source d'un composant. C'est souvent un avantage mis en avant dans les domaines où la propriété intellectuelle est importante.

Si l'on souhaite exécuter un système sur des cibles hétérogènes et donc limiter les hypothèses concernant les futures cibles d'exécution du système, un langage de programmation classique est préférable. Par contre, une phase de compilation doit être réalisée afin de pouvoir utiliser le composant. L'installation ou le remplacement de composants pendant l'exécution est alors plus difficile.

Déploiement :
Le déploiement correct d'un composant nécessite plusieurs informations supplémentaires. Celles-ci peuvent être générales, comme le nom de l'auteur ou le numéro de version ; être utiles à l'installation, comme l'arborescence des fichiers ou encore les librairies requises. Elles peuvent également être composées des besoins extra-fonctionnels nécessaires au déploiement, comme le besoin de persistance ou de sécurité, par exemple.

Analyse :
Les deux premiers points se focalisent sur la problématique d'exécution d'un composant. Cependant, si l'on désire effectuer des analyses pour s'assurer de la correction d'un composant primaire ou composite, les deux techniques précédentes ne sont pas adaptées. Il est alors nécessaire de fournir les renseignements sur le comportement de composants afin de permettre d'effectuer des analyses. Ces renseignements sont utilisés soit sur un composant, soit sur un assemblage de composants, soit lors de la connexion d'un composant à un autre composant (chargements à l'exécution), soit lors de son déploiement ou encore lors de son exécution (contrôles). Un des points importants et non trivial est bien sûr de s'assurer que le comportement ainsi décrit de manière abstraite reflète l'exécution du composant.

L'énoncé des propriétés, aboutissant à un comportement en boîte grise, peuvent être diverses et variées selon le contexte. On trouve, par exemple, les propriétés temporelles, les types de cibles pour lesquelles le composant est prévu, le nom de l'algorithme implémenté, le numéro de version, le cycle de vie, etc. Il faut noter que la plupart de ces propriétés sont liées au comportement du composant à l'exécution. À l'image de la pire ou de la meilleure durée d'exécution (respectivement " *Worst/Best Case Execution Time*" ou WCET/BCET), les propriétés peuvent être une contrainte portant sur l'implémentation du composant auquel cas on parle de budgets temporels (CLARA [40], EAST-EEA [101]) ou le résultat de l'analyse de son exécution (MetaH [125], VEST [112]).

Nous avons vu les différentes façons de décrire le comportement d'un composant. Nous allons maintenant voir les différentes techniques utilisées pour accéder à ce comportement.

2.2.3.2 Les interfaces

Une interface est un point d'interaction d'un composant avec son environnement. Dans la plupart des approches, ces points d'interactions sont orientés. On trouve des interfaces en entrée ou en sortie (PBO [113], rubus [63], giotto [61]) ou des interfaces fournies ou requises (EAST-EEA [101], koala [124], Think [44], UML2.0 [97]). Classiquement les interfaces en entrée ou en sortie sont utilisées lorsque la communication est de type flot de données ou événementielle alors que les interfaces fournies ou requises servent aux communications de type client/serveur.

Les interfaces peuvent être classées suivant le type d'interaction qu'elles implémentent. En se basant sur les approches existantes, on discerne trois principaux types d'interfaces :

1. Les interfaces fonctionnelles qui permettent d'accéder à un service particulier d'un composant ;
2. les interfaces de configuration qui permettent d'ajuster certains paramètres relatifs au comportement d'un composant (choix de l'algorithme, etc.) ;
3. les interfaces de synchronisation qui permettent de déclencher ou de suspendre l'exécution d'un composant.

Puisque les interfaces sont les seuls points de communication d'un composant avec son environnement, elles représentent la spécification du composant. Cette spécification doit idéalement

permettre d'utiliser le composant en boîte noire. Afin d'atteindre cet objectif, [15] propose de décrire les spécifications d'une interface selon quatre niveaux de contrats distincts :

Contrat de niveau 1 il spécifie la signature des services offerts ou requis par un composant. Ce niveau de contrat peut être établi en mettant en correspondance un ou plusieurs champs textuel(s) décrivant la signature d'un service.

Contrat de niveau 2 il spécifie les contraintes sur l'utilisation d'un service. Ces contraintes sont essentiellement réalisées à l'aide d'assertions booléennes (pré et post conditions, invariants). Ces contraintes peuvent être décrites en OCL [96] mais également être spécifiées à l'aide d'un algèbre de processus [2] ou de manière plus complexe à l'aide d'un méta-modèle du rôle exercé par le service spécifié [69].

Contrat de niveau 3 il spécifie le protocole de synchronisation entre les appels aux différents services composant le comportement d'un composant. Cette synchronisation peut être exprimée par des champs textuels comme une séquence ou "aléatoire" ou de manière plus complexe par des automates spécifiques de type "*protocol statechart*" [75].

Contrat de niveau 4 il spécifie les caractéristiques extra-fonctionnelles que l'on désire observer sur le service associé. Il exprime le niveau de QoS (*Quality of Service*) requis ou offert pour/par une interface comme dans [123, 64]. À ce titre le contrat de niveau 4 est aussi appelé contrat de QoS. Globalement, dans le domaine de l'embarqué, rares sont les approches qui intègrent ce niveau de contrat.

Conclusion sur les interfaces Les interfaces sont les seuls points de communication entre un composant et son environnement. Elles permettent de spécifier comment interagir avec un composant, tant au niveau fonctionnel qu'extra-fonctionnel.

Si un composant doit être utilisé par un tiers dans un environnement a priori non connu, alors la description de l'interface devra être assez précise pour éviter une mauvaise utilisation du composant. Ce dernier peut alors être vu comme une entité de réutilisation auto-cohérente (*self consistent*), utilisable en boîte noire (cf. définition d'un composant par Szyperski [116]).

A contrario, si l'utilisation d'un composant est confinée à un type de configuration connue et a priori fixe, la description d'une interface peut être réduite. La réutilisation du composant nécessite alors une connaissance minimale de son comportement. Afin d'assurer un fonctionnement correct, l'utilisation (ou "composabilité") du composant est d'une part liée à une utilisation correcte de son interface et d'autre part à l'analyse de son assemblage au sein de la configuration.

De nombreuses approches permettent de vérifier la conformité sémantique et syntaxique (contrats de niveau 1 et 2) des interfaces lors de leur connexion. Quelques approches s'intéressent aux contrats de niveau 3. Cependant, les contrats de QoS (de niveau 4) sont souvent mis de côté. Lorsqu'ils sont adressés, les approches proposent généralement un langage permettant la description de contrat de QoS et/ou propose une architecture permettant la négociation de contrat à l'exécution. La sémantique de satisfaction permettant l'établissement des contrats de niveau QoS est alors simplement basée sur un opérateur de comparaison de quantité de ressource utilisée. Ces préoccupations sont pourtant nécessaires à l'établissement d'une architecture possédant des contrats de QoS.

Nous venons de présenter le rôle d'une interface ainsi que la notion de contrat, nous présentons maintenant la manière dont ces interfaces peuvent être interconnectées.

2.2.3.3 Les connecteurs

Un connecteur est en charge des interactions entre les composants d'une architecture. Il spécifie les règles qui régissent la relation entre deux ou plusieurs composants via leurs interfaces. Il réalise donc l'établissement de chacun des contrats décrits dans le paragraphe 2.2.3.2.

Certaines approches considèrent les connecteurs comme de simples liens logiques (MetaH [125], giotto [61], PECOS [57], PBO [113], rubus [63], koala [124]). Ces approches permettent alors de n'établir des contrats qu'entre des interfaces homogènes.

Dans d'autres approches, le connecteur peut posséder un comportement appelé "glue" (Wright [2], unicon [108], Think [44], VEST [112], BIP [10]). Il a pour objectif l'adaptation des interfaces afin de permettre l'établissement de contrats entre des interfaces hétérogènes.

Un connecteur spécifie également le type de communication entre composants. Par exemple, un connecteur peut spécifier si les échanges de données se font par tunnels (*pipe*) ou par mémoire partagée (PECOS [57], PBO [113], rubus [63]). D'autres propositions permettent de les utiliser pour spécifier le type de synchronisation lors de la mise à jour d'une donnée (rendez-vous, section critique, etc.) (CLARA [40], Wright [2], BIP [10]).

Lorsque le connecteur possède un comportement, la "glue" peut soit être décrite de la même manière que le comportement d'un composant (primaire ou composite) comme dans Wright [2], unicon [108], Think [44], VEST [112], Wright [2], soit de manière spécifique comme dans BIP [10] ou [20]. La "glue" peut également être choisie parmi une liste finie de comportements (CLARA [40]).

La description de la "glue" peut être plus ou moins éloignée des détails de son implémentation. En particulier, [20] propose l'utilisation de modèles (diagramme de collaboration, contraintes OCL, etc) pour décrire la "glue" d'un connecteur. Cette description est ensuite rafinée et peut aboutir à différentes implémentations selon les critères choisis (performances, sécurité, etc).

Conclusion sur les connecteurs Les utilisations des connecteurs énoncés précédemment ne sont que des exemples. Les connecteurs sont très hétérogènes suivant les approches. On trouve ainsi différents types de connecteurs allant du connecteur logique (assimilable à un fil) jusqu'au connecteur complexe décrit par une configuration (un assemblage de composants).

La complexité d'un connecteur est encore une fois liée au domaine d'application. Les connecteurs simples sont, de préférence, réservés aux systèmes ayant des besoins d'installations et/ou de remplacements de composants pendant l'exécution. Ils peuvent également être utilisés pour figer la sémantique de communication entre les composants, facilitant, de fait, la réalisation d'analyse. Cependant, les connecteurs logiques ne peuvent être utilisés que lorsque les composants de la configuration forment un ensemble homogène sur les quatre niveaux de contrat des interfaces.

A contrario, les connecteurs complexes sont utilisés dans les systèmes où les besoins de spécification en termes de communication/synchronisation sont plus forts. En particulier, ils permettent de faire communiquer des composants hétérogènes en termes de signature (contrat de niveau 1), de comportement (contrat de niveau 2) et de protocole de synchronisation (contrat de niveau 3).

À notre connaissance, aucune approche ne propose de connecteurs permettant l'établissement de contrats de QoS lors de la composition d'interfaces.

Grâce aux connecteurs, nous avons vu comment relier les interfaces des composants. Le résultat d'un assemblage est appelé configuration. La section suivante introduit cette notion.

2.2.3.4 La configuration et son niveau de description

Une configuration est un graphe bipartite de composants et de connecteurs. C'est le résultat d'une description architecturale pour un système donné. À ce titre, une configuration est aussi communément appelée "architecture" dans la littérature.

Plusieurs types de configuration existent suivant la place qu'elle occupe dans le cycle de vie du système. La littérature identifie principalement trois classes de configuration dont les noms peuvent varier. Nous prenons dans cette thèse la terminologie suivante :

Configuration ou architecture logique : C'est une vue architecturale des aspects fonctionnels d'un système sans prise en compte de la plateforme de déploiement. Le niveau d'abstraction de cette configuration peut aller d'une description des relations entre les différentes entités composant le système jusqu'à la spécification du comportement, voire la mise en œuvre des algorithmes utilisés. Les aspects extra-fonctionnels d'un système sont souvent liés au matériel sur lequel le système s'exécute. Il est donc difficile de les incorporer à la description d'une architecture logique. Toutefois, pour les aspects temporels, CLARA [40] et EAST-EEA [101] proposent l'utilisation de budget temporel. Un budget temporel est l'estimation d'un temps (WCET, temps de blocage, etc.) servant de contrainte lors du raffinement de l'architecture logique ou lors de son déploiement.

Configuration ou architecture technique/support : C'est une vue architecturale de la plateforme de déploiement. Cette spécification peut être réalisée à des niveaux différents. Elle peut être une description du matériel sur lequel s'exécute le logiciel (processeurs, bus, mémoire, etc.) aussi bien qu'une description des services offerts par le système d'exploitation de la plateforme (service d'accès aux tâches, aux media de communication, etc.). À ce niveau, certaines informations extra-fonctionnelles sur la plateforme telles que la vitesse du processeur ou la taille de la mémoire peuvent être renseignées (VEST [112], EAST-EEA [101]).

Configuration ou architecture opérationnelle : C'est une vue architecturale résultant du déploiement de l'architecture logique sur l'architecture technique. Elle décrit donc le système complet, avec les tâches, les protocoles de communication, la politique d'ordonnancement, etc. Il est alors possible de donner une description réelle des aspects extra-fonctionnels du système. En particulier, les informations sur le comportement temporel,

comme par exemple le temps de réponse, sont des valeurs mesurées une fois le déploiement réalisé.

Conclusion sur les configurations Pour les systèmes embarqués temps réel, la majorité des ADLs se concentrent sur la description d'une configuration opérationnelle ou confondent la configuration logique et la configuration opérationnelle (MetaH [125], giotto [61], COTRE [14], PECOS [57], PACC [128], PBO [113], rubus [63], AADL [99]). Cette vue architecturale permet alors de s'assurer du respect des contraintes fonctionnelles et extra-fonctionnelles. Pour les systèmes visés, travailler sur la configuration opérationnelle permet de réaliser, entre autres, les analyses classiques du domaine visé comme l'analyse d'ordonnancement temps réel (MetaH [125], giotto [61], COTRE [42], rubus [63]), l'évaluation des temps de réponse (MetaH [125], COTRE [42]) ou le calcul de l'empreinte mémoire (Koala [124]).

On trouve cependant quelques approches qui identifient clairement la configuration logique (CLARA [40], EAST-EEA [101], VEST [112], koala [124]). Cette vue architecturale permet la réalisation de premières analyses destinées à valider une configuration logique candidate et ainsi s'assurer de la cohérence de ses descriptions extra-fonctionnelles. Une fois la description de l'architecture technique réalisée, des analyses doivent permettre de s'assurer que les résultats obtenus lors des analyses de la configuration logique sont préservés lors du déploiement (CLARA REACT [45], EAST-EEA [101]).

Il semble intéressant de séparer la configuration logique de la configuration opérationnelle afin de séparer les préoccupations au niveau fonctionnel mais également extra-fonctionnel. En effet, les analyses réalisées sur une configuration logique permettent de détecter les incohérences extra-fonctionnelles pouvant exister et assurer une dérivation correcte des contraintes extra fonctionnelles. On sépare ainsi la faisabilité d'une description architecturale de sa réalisation sur une configuration technique particulière.

Chacun des niveaux de configuration spécifie différents points de vue sur la structuration du système. L'étude des architectures logicielles montre que certaines structurations sont récurrentes et utilisées pour résoudre une classe spécifique de problèmes.

2.2.3.5 Le style architectural

Les contraintes permettant d'obtenir ces structurations particulières ont été identifiées et classées comme "styles architecturaux" (*architectural style*). Parmi les définitions existantes autour de cette notion, nous considérons celle-ci :

"*An architectural style is a coordinated set of architectural constraints that restricts the role / features of architectural elements and the allowed relationships among those element within any architecture that conforms to that style*" [48].

Cette définition datant de 2000 signifie qu'un style architectural est un ensemble de contraintes permettant de limiter le rôle, le comportement ainsi que la manière d'assembler les éléments

(composants et connecteurs) des configurations appliquant ce style. D'après [54], les contraintes définies par un style architectural permettent de :

- définir un vocabulaire fini de type de composants et de connecteurs ;
- définir des contraintes sur les associations entre ces entités ;
- définir une interprétation sémantique pour chacun des éléments utilisés dans la réalisation d'une configuration ;
- définir les analyses pouvant être réalisées sur les configurations appliquant ce style.

Classiquement, on identifie en génie logiciel les styles architecturaux suivant : "*Pipe and filter*", "*Blackboard*", "*Object-oriented*" and "*Layered systems*" [109]. Il en existe bien sûr d'autres et chacun de ces styles possède des variantes. De plus certains styles sont le résultat d'un mélange de plusieurs styles.

L'utilisation ou la définition d'un style architectural, en particulier l'énonciation de contraintes, est souvent motivée par l'application d'un principe de génie logiciel [58]. En effet, [70, 71] mettent en avant le fait que ces différentes contraintes entraînent un ensemble de propriétés de qualité, au sens du génie logiciel, sur la configuration qui les applique.

On distingue principalement deux types de propriétés extra-fonctionnelles : les propriétés chiffrables et les propriétés non chiffrables. Les propriétés non chiffrables sont par exemple : l'efficacité, la facilité d'évolution et de réutilisation. Elles sont souvent désignées par le terme "attributs de qualité" (*quality attribute*) [9]. De nombreuses études, parmi lesquelles [121, 68, 67], se focalisent sur leur évaluation.

Selon les propriétés extra-fonctionnelles recherchées, un style architectural est plus ou moins efficace. Le tableau suivant, en partie extrait de [31] donne un aperçu du comportement de chaque style vis-à-vis de cinq propriétés extra-fonctionnelles non chiffrables. Il est possible de modérer

architectural style	Performance	Maintainability	Reliability	Safety	Security
Pipes & filters	+-	+-	-	-	+
Blackboard	-	+	+-	-	+-
Object-Oriented	+-	+-	+-	+	+-
Layered	-	+	+-	??	??

+- : non pertinent, + : effet positif, - : effet négatif, ?? : non déterminé.

TAB. 2.1 – Comparaison des styles architecturaux classiques vis-à-vis de propriétés extra-fonctionnelles non chiffrables.

l'impact d'un style sur un attribut de qualité en réalisant une variante d'un style particulier ou en utilisant un style hétérogène. Un style hétérogène possède les propriétés de chacun des styles qu'il emploie. Les styles employés peuvent être combinés de plusieurs manières. On peut, par exemple, les combiner de manière hiérarchique. Ainsi une architecture en couche (*Layered*) peut, à l'intérieur de chacune des couches utiliser un style *Pipe and filter*. Chaque composant du style *Pipe and filter* peut être décrit grâce à un style *Objet oriented* [109].

Pour leur part, les propriétés extra-fonctionnelles chiffrables peuvent être liées aux aspects temporels et de performance. Certains styles architecturaux, en imposant un langage formel et une structure particulière facilite l'évaluation de ces propriétés [54]. Un style architectural adapté doit donc définir des types de composants et de connecteurs dont le comportement permet l'ana-

lyse de la propriété recherchée. Il peut également définir certaines contraintes sur les assemblages de composants et de connecteurs afin d'assurer que la configuration est analysable vis-à-vis de la propriété recherchée. Par exemple, dans l'embarqué, pour assurer la possibilité de faire une analyse du temps de réponse, un style architectural adéquat doit imposer la spécification des éléments nécessaires à cette analyse (par exemple, des tâches périodique pour faire une analyse RMA [72]). Il faut donc noter qu'un style architectural impose la spécification des éléments nécessaires pour effectuer une analyse mais ne peut ni spécifier la manière de réaliser cette analyse, ni assurer du résultat de l'analyse.

Conclusion sur le style architectural La littérature portant sur les architectures logicielles a mis en avant l'importance du style architectural puisque son application permet non seulement d'imposer certains principes de génie logiciel, mais aussi d'imposer une structure analysable.

Pour autant, comme le souligne [48], il n'existe pas de style architectural "miracle" (*"silver-bullet"*). Chaque style possède ses avantages et ses inconvénients. C'est alors à l'architecte de faire ou d'appliquer le style favorable à son problème.

Il faut faire une différence franche entre l'évaluation de propriétés extra-fonctionnelles liées à un style architectural et la propension qu'apporte un style architectural à l'évaluation de propriétés extra-fonctionnelles.

D'un côté l'application d'un style architectural peut fournir des propriétés extra-fonctionnelles non chiffrables. On peut par exemple dire qu'un style architectural " X " permet d'obtenir une bonne réutilisabilité sur les configurations qui l'appliquent.

D'un autre côté on ne peut pas assurer des propriétés extra-fonctionnelles chiffrables ; c'est-à-dire qu'il n'est pas possible de dire que le style architectural " X " permet d'obtenir un temps de réponse de " a ". En revanche, on peut dire que la propriété est évaluable.

Enfin, il est intéressant de remarquer que les styles architecturaux, puisqu'ils spécifient des conditions d'utilisation sur des éléments du langage de description (composants, connecteurs, etc.) sont intimement liés à ce langage. Tout langage impose donc de manière implicite un style architectural. De la même manière, un modèle à composant, avec des contraintes de composition fortes, peut définir implicitement un style architectural [32, 127].

Une fois les types de composants, de connecteurs et les contraintes sur leurs assemblages identifiés, il est nécessaire de les exprimer dans un langage particulier.

2.2.3.6 Le langage

Il existe de nombreux langages permettant de décrire une configuration. On trouve notamment deux grandes familles de langages : les langages de description d'architecture (ADLs) (MetaH [125], giotto [61], COTRE [14] , AADL [99], CLARA [40], EAST-EEA [101], VEST [112], unicon [108], Wright [2]) et les modèles à composants (notés CMs pour *Component Model*) (PECOS [57], PACC [128], PBO [113], rubus [63], koala [124], Think [44], osgi [98] , .Net [86], EJB

[88]).

La multiplication des langages provenant de ces deux familles a introduit de nombreux formalismes et notations (textuelles, graphiques ou mixtes). Aujourd'hui, du fait d'UML, les langages ont tendance à s'unifier. En particulier, UML2 [97] permet de prendre en charge la description de composants, d'interfaces et de connecteurs, soit les éléments nécessaires pour décrire des architectures. De plus, UML2 permet de modéliser le système sous développement à plusieurs niveaux. Par exemple, les composants UML2 possèdent une vue boîte noire de par la spécification d'interfaces requises et fournies mais également une vue boîte blanche qui est, quant à elle, cachée à l'utilisateur du composant. Cette vue s'appuie sur un ensemble de *realizations* implémentant le comportement du composant. Une fois une configuration spécifiée, UML2 permet également de spécifier la phase de déploiement. Durant cette phase, un composant UML2 est représenté par un ou plusieurs *artifacts*. Un *artifact* UML2 permet de modéliser ce que l'on veut produire à partir d'un modèle, de manière indépendante des modèles. Ainsi, un même modèle peut produire certains *artifacts* pour les tests locaux, d'autres pour l'intégration, et d'autres pour un portage sur d'autres environnements. .

Puisque les entités décrites dans UML2 sont génériques, des points de variations sémantiques sont introduits [106]. Pour pallier à ces "trous" sémantiques, certains langages de description d'architecture comme AADL [99] proposent une spécialisation des entités d'UML2 pour définir leur langage. Ainsi AADL ajoute des entités spécifiques à celles décrites dans UML2. AADL se focalise alors sur la description d'entités de bas niveau fortement liées à l'implémentation. Un des points intéressant d'AADL réside dans le fait qu'il est l'un des premiers langages à permettre l'ajout de QoS dans la spécification des interfaces d'un composant.

Enfin, il faut noter qu'un style architectural peut être défini par la formulation d'un langage spécifique (DSSA : *Domain-Specific Software Architecture*) ou par la spécialisation d'un langage existant. Aesop [54] est un langage de description d'architecture spécifique permettant la définition d'un style architectural. De sont côté, UML permet l'expression d'un style architectural au travers de deux mécanismes :

- il permet d'être étendu pour la création de types de composants particuliers (notion de spécialisation, profils, etc.) ;
- il peut être contraint de différentes manières comme par l'utilisation de OCL (*Object Constraint Language*) [96] ou par la définition d'un méta-modèle [50].

Définir un langage spécifique, et donc un style architectural, à l'aide d'un méta-modèle permet de définir l'ensemble des concepts utilisés. Ces concepts peuvent alors être utilisés pour définir un environnement de modélisation associé aux concepts du style architectural. Ce sont les principes utilisés par les outils EMF [18], MetaEdit+ [122], GME [62] et très récemment dans TopCased [43]. Ceux-ci permettent d'obtenir rapidement un environnement de développement à partir de la description des concepts utilisés.

2.2.3.6.1 Conclusion sur le langage De très nombreux formalismes et notations ont vu le jour ces vingt dernières années. Cela entraîne une difficulté de compréhension des notations, grammaires et des avantages/inconvénients liés à chacun des langages. On distingue deux grandes familles parmi ces langages : les ADLs et les CMs. Les ADLs, sont historiquement des langages qui permettent de spécifier une abstraction du comportement désiré d'une configuration pendant

l'exécution. Contrairement à cela, les CMs décrivent des langages qui permettent d'encapsuler proprement le code d'un composant afin de pouvoir les développer indépendamment les un des autres.

Au niveau des formalismes, on voit de plus en plus d'ADL basés sur UML (AADL [99], EAST-EEA [101], VEST [112]) ; UML ayant l'avantage de fournir une notation de plus en plus universelle et des outils existants. Cependant, au-delà des notations, l'essentiel des efforts portent aujourd'hui sur les concepts manipulés. Dans ce contexte l'ingénierie dirigée par les modèles offre des outils de manipulation des concepts au travers de l'utilisation de méta-modèles et de transformations de modèles.

De manière plus générale, l'ingénierie dirigée par les modèles offre un cadre méthodologique et technologique qui permet d'unifier différentes façons de faire dans un processus homogène. Il est ainsi possible d'utiliser la technologie la mieux adaptée à chacune des étapes du développement, tout en ayant un processus global de développement qui est unifié dans un paradigme unique [65]. Ainsi, il nous parait clair que l'utilisation de modèles pour la description d'une architecture est, aujourd'hui, l'approche la plus appropriée.

Il faut être conscient que l'utilisation d'un langage particulier impose souvent l'utilisation du style architectural sous jacent. Le style est alors utilisé de manière implicite. Seuls certains langages comme C2 [83] fournissent explicitement le style architectural qu'ils utilisent.

Pour assurer la correction des configurations décrites, le langage utilisé se doit de posséder une sémantique opérationnelle formelle et adaptée. Le langage utilisé doit donc fournir une sémantique opérationnelle en adéquation avec les analyses à réaliser. Ces analyses sont le sujet du paragraphe suivant.

2.2.4 Analyse d'une configuration

Les analyses basées sur les architectures logicielles permettent de s'assurer de la correction d'une configuration vis-à-vis d'une propriété donnée.

Certaines configurations, pouvant être modifiées dynamiquement lors de l'exécution, permettent de vérifier des propriétés pendant l'exécution du système (Qinna [123], osgi [98], .Net [86], EJB [88]). Ces configurations ont alors besoin de pouvoir identifier les composants pendant l'exécution du système. À l'inverse, les systèmes visés par l'étude ont des besoins de validation a priori, ce chapitre se concentre donc sur les analyses réalisées lors du développement du système.

Comme précisé dans le chapitre 2.2.3.5, il existe deux grandes catégories de propriétés vérifiables. La première concerne les propriétés qualitatives (non chiffrables) comme le niveau de réutilisabilité de l'architecture ou sa facilité de maintenance. Les études de ces propriétés, parmi lesquelles [121, 68, 67], sont basées sur des techniques non formelles et non automatisables (cf. analyse de [126]).

La deuxième catégorie concerne les propriétés quantitatives (chiffrables) comme les propriétés temporelles, de performances, etc. Ces propriétés sont essentielles à la validation des systèmes

temps réel.

Les travaux décrits dans [30] proposent une classification des propriétés extra-fonctionnelles liées aux composants et à leur configuration. Cette classification fait émerger deux classes de propriétés clairement différenciables :

Les propriétés directement composables (*directly composables properties*) : ce terme désigne les propriétés d'une configuration qui sont fonction d'un et un seul type de propriétés de chaque composant appartenant à la configuration. La propriété de l'assemblage est alors du même type que la propriété de chaque composant.

Les propriétés dérivées (*derived properties*) : également appelées propriétés émergentes ; ce terme désigne les propriétés d'une configuration qui dépendent de plusieurs propriétés des composants appartenant à la configuration. Ces propriétés peuvent ou non être réifiées au niveau du composant.

Parmi les propriétés dérivées dans le domaine des systèmes temps réel, on trouve notamment les analyses des propriétés temporelles suivantes :

vivacité : quelque chose de bien arrivera un jour, COTRE [42], CLARA [45], EAST-EEA [101], MetaH [125] ;

sûreté : quelque chose de mauvais n'arrive jamais, MetaH [125], COTRE [42] ;

respect des échéances : le temps écoulé entre deux événements dans le système est inférieur ou égal à un temps donné, COTRE [42], CLARA [45], EAST-EEA [101], MetaH [125].

Ces analyses, ainsi que celles réalisées dans [78, 126], se basent sur une description formelle et abstraite du comportement d'un composant à l'aide d'automates temporisés (MetaH [125], COTRE [42], CLARA [40]). L'évaluation est alors réalisée par des outils basés sur les techniques formelles tels que UPPAAL [76], Kronos [130] ou CADP [46].

On trouve également, parmi les propriétés dérivées, les analyses d'ordonnançabilité (MetaH [125], giotto [61], PACC [128], COTRE [42], rubus [63], VEST [112], Unicon [108]). Ces analyses nécessitent la connaissance du WCET de chaque composant, leur période d'activation, le modèle de tâches sous-jacent, la politique d'ordonnancement ainsi que les différentes synchronisations entre les composants.

Sous certaines conditions, il est possible d'analyser certaines propriétés dérivées sur des sous parties d'une configuration. Par exemple, l'analyse d'ordonnançabilité d'un système distribué sur plusieurs unités de calculs peut être analysée unité par unité pour chaque configuration partielle regroupant les composants et les connecteurs s'exécutant sur l'unité de calcul considérée.

On trouve également dans la littérature la vérification de propriétés directement composables comme l'empreinte mémoire réalisée par Koala [124]. Ces propriétés peuvent être vérifiées lors de chaque connexion d'un composant avec le système. Ces analyses peuvent être réalisées au fur et à mesure de la construction d'une configuration et servent ainsi d'aide à la construction du système.

On trouve enfin des approches pour la vérification de la conformité à un style architectural particulier. Ces analyses permettent de déceler rapidement les erreurs structurelles pouvant avoir un impact sur la configuration globale (utilisation d'un type de composant non analysable, etc.). Les contraintes sont alors exprimées à l'aide d'un langage de contraintes tel que OCL [96], d'un méta-modèle [50] ou d'un artefact spécifique à l'ADL comme dans [54, 2].

Conclusion sur les analyses

Puisque les analyses de propriétés qualitatives ne donnent pas de résultats chiffrables, elles peuvent uniquement servir de base de comparaison entre différentes configurations candidates. Elles sont indépendantes d'un ADL particulier et permettent de comparer des configurations décrites à l'aide de divers ADLs. Ces méthodes d'analyses sont étroitement liées à l'analyse des impacts d'un style architectural particulier [70]. Elles ne donnent pas d'informations sur l'exactitude de la configuration réalisée mais sur des propriétés de qualité au sens du génie logiciel (maintenabilité, réutilisabilité, etc.).

Parmi les analyses quantitatives comme celles des propriétés temporelles on peut différencier deux grandes familles : les propriétés directement composables et les propriétés dérivées (ou émergentes). Les informations permettant d'évaluer chacune de ces propriétés peuvent être encapsulées dans des interfaces. L'établissement d'un contrat de QoS lors de la connexion de deux interfaces est possible uniquement pour des propriétés directement composables. Les propriétés émergentes ne peuvent pas directement donner lieu à l'établissement d'un contrat de QoS puisqu'elles nécessitent l'analyse d'un assemblage de composants. Cependant, une fois la configuration analysée, le résultat de l'analyse peut être utilisé pour établir le contrat.

Enfin, pour pouvoir faire des analyses de propriétés extra-fonctionnelles et chiffrables, une sémantique opérationnelle doit être fournie pour chacun des éléments architecturaux mais aussi pour chacune des propriétés impliquées dans l'analyse. Dans le domaine des systèmes temps réel, les automates temporisés et les outils associés sont souvent utilisés pour abstraire le comportement et fournir une sémantique opérationnelle temporisée.

2.2.5 Mise en œuvre d'une architecture logicielle

Une fois la configuration d'un système réalisée et validée, la phase finale est sa mise en œuvre. On distingue principalement deux stratégies :

1. La projection du code exécutable de chaque composant et de chaque connecteur directement sur une infrastructure permettant son exécution (cf. figure 2.3) ;
2. La compilation de la configuration vers un code exécutable et monolithique de la configuration entière (cf. figure 2.4).

Dans le premier cas, l'exécution devra être gérée par un intergiciel spécifique permettant l'installation et l'exécution du code exécutable de chaque entité architecturale. Les composants restent alors identifiables à l'exécution. Dans ce cas, on trouve deux approches pour gérer la communication entre les composants. Si les connecteurs ne sont qu'une vue logique (cf. 2.2.3.3) ou qu'ils ne peuvent être choisis que parmi une librairie de connecteurs prédéfinis, la communication entre les composants est réalisée par l'intergiciel (cf. figure 2.3 B). Lorsqu'au contraire les connecteurs sont des entités pouvant être décrites de manière plus complexe, le code exécutable de la "glue" du connecteur est exécuté par l'intergiciel, comme pour un composant (cf. figure 2.3 A).

Dans la deuxième approche, la configuration est compilée dans un langage intermédiaire compilable/exécutable sur la cible de déploiement (configuration technique). Les composants ne

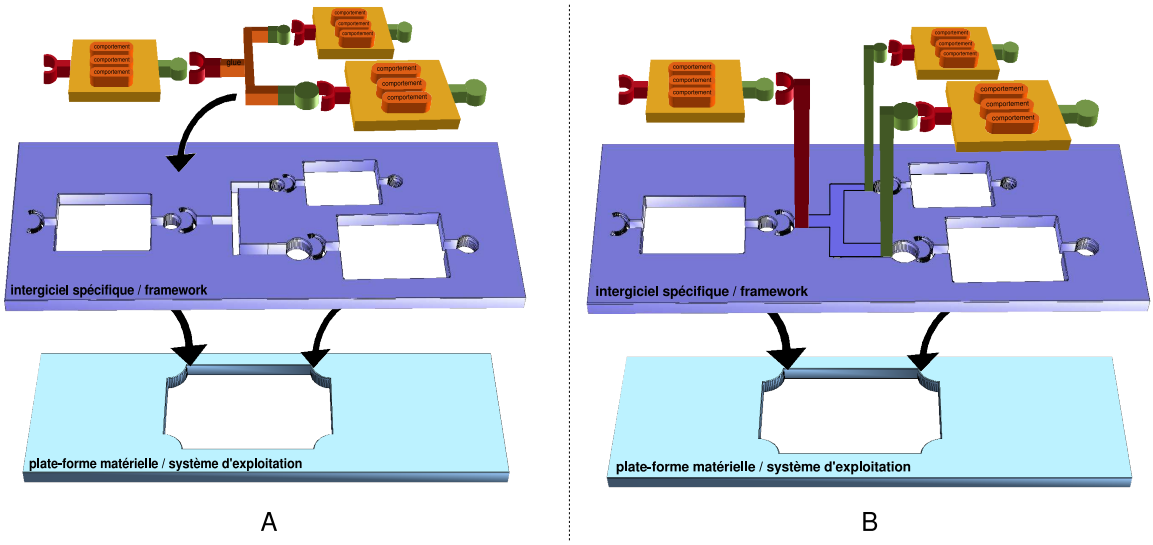


FIG. 2.3 – Deux mises en œuvre d'une configuration basée sur un intergiciel

sont plus réifiables ; le découpage en composants de la vue architecturale a disparu à l'exécution.

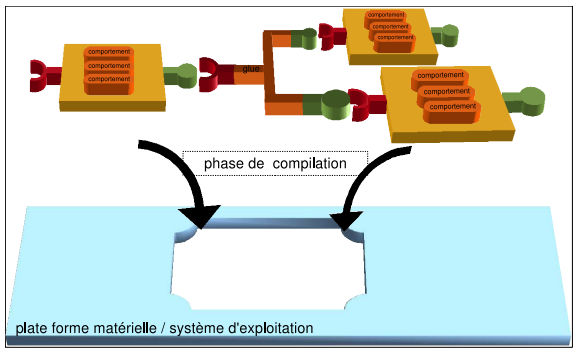


FIG. 2.4 – l'implémentation d'une architecture logicielle compilée

Conclusion sur la mise en œuvre d'une configuration

En règle générale, le choix d'une de ces stratégies est lié à la stratégie choisie lors de la description du comportement des composants. Si le comportement est décrit sous forme binaire ou de *byte code*, son exécution est réalisée par un intergiciel. Dans le cas d'une description du comportement sous forme de code source (formel ou non) le choix reste libre, même si généralement la configuration est compilée. Ce choix est important et intervient très tôt dans le développement.

L'utilisation d'un intergiciel permet de conserver la séparation en composants pendant l'exécution. Elle permet ainsi de contrôler l'exécution, l'arrêt, le remplacement, la suppression et la reconfiguration de chacun des composants. Cependant, le modèle d'allocation des ressources

d'exécution (tâches) sous-jacent est souvent préétabli et non optimisé (par exemple une tâche par composant).

La compilation d'une configuration permet d'obtenir un code plus efficace en termes d'empreinte mémoire et plus performant pour l'allocation des ressources. La configuration logique peut même être orthogonale au modèle de tâche choisi et donc à la configuration opérationnelle. Des études comparatives montrent alors l'impact de tels choix vis-à-vis de critères comme le respect des échéances ou l'utilisation des ressources [60, 51]. C'est donc, pour ces raisons que la compilation d'une configuration est la solution la plus fréquemment retenue dans le domaine des systèmes embarqués temps réel.

2.2.6 Conclusion sur les architectures

Dans ce chapitre, nous avons présenté les différents éléments nécessaires à la description d'une architecture logicielle. Une architecture logicielle est définie par une **configuration** ; c'est-à-dire par un ensemble de **composants** interconnectés. Chaque composant est défini par un **comportement** décrit de manière plus ou moins abstraite par rapport à l'implémentation. Ce comportement n'est accessible qu'au travers d'**interfaces**. Enfin les interfaces des composants sont interconnectées à l'aide de **connecteurs**. Un **ADL** est un langage permettant d'exprimer chacun de ces éléments.

Nous avons vu qu'il existe de nombreuses possibilités pour décrire les éléments d'une architecture logicielle. En particulier, il existe de nombreux langages de description architecturale. Cependant, l'utilisation de modèles (et de méta-modèles), en se concentrant sur la manipulation des concepts, nous apparaît comme fédérateur. Cette approche permet en effet de définir les concepts d'un domaine plutôt que leur syntaxe. Travailler au niveau des concepts permet également d'effectuer des transformations de ces concepts vers d'autres concepts, par exemple dans un but de raffinement. Au niveau de la mise en œuvre des modèles, UML s'impose comme un standard de fait et propose de nombreux outils.

Pour ce qui est de la définition du comportement exécutable et de son exécution, les architectures logicielles pour les systèmes embarqués utilisent encore largement le langage C. La description architecturale est alors compilée et s'exécute généralement directement sur un système d'exploitation temps réel. Cependant, l'introduction d'une surcouche logicielle au système temps réel permet la mise en œuvre de certains mécanismes récurrents comme, par exemple, la communication entre les composants. Elle permet également de s'abstraire d'une cible particulière. Bien sûr, l'utilisation d'un intergiciel se fait au détriment de l'utilisation des ressources, qui s'en trouve augmentée.

Nous avons vu que les services proposés par un composant sont accessibles via des interfaces en entrées/sorties ou offertes/requises. Ces interfaces permettent de donner une spécification du composant auquel elles sont associées via quatre niveaux de contrats. Ces contrats permettent de s'assurer de l'utilisation correcte d'un composant vis-à-vis de sa spécification fonctionnelle et extra-fonctionnelle. En particulier, l'établissement de contrats de qualité de services est nécessaire à l'utilisation de composants dans le domaine d'application visé où les contraintes extra-fonctionnelles sont fortes.

Une attention particulière est apportée aux choix permettant de réaliser l'analyse de tels systèmes. Ces analyses concernent principalement l'utilisation des ressources (empreinte mémoire, ordonnancement, etc.) et les contraintes temporelles (échéance de bout en bout, etc.). L'utilisation des ressources et les contraintes temporelles peuvent être vérifiées lors de l'exécution d'un système ou durant sa spécification. Puisque les systèmes de contrôle de processus peuvent posséder un comportement critique en cas de dysfonctionnement, il est préférable de réaliser les analyses pendant la spécification du système.

On distingue deux types d'analyses différentes :

1. Les analyses réalisées sur une configuration : une vue boîte grise du comportement des composants et des connecteurs est alors nécessaire. Cette vue boîte grise doit permettre d'évaluer l'impact de la configuration sur ses propriétés extra-fonctionnelles. Enfin, il faut remarquer que si le comportement est abstrait dans un langage particulier, permettant la réalisation d'analyse, la sémantique opérationnelle de la future mise en œuvre doit être équivalente à celle du langage utilisé lors de l'analyse. L'utilisation de ce type d'analyse est souvent réservée à l'analyse des propriétés émergentes d'un système.
2. L'établissement d'un contrat de QoS lors de la connexion entre deux interfaces : le connecteur doit permettre d'assurer l'établissement de ce contrat. Cela permet de s'assurer d'une composition correcte vis-à-vis des contraintes extra-fonctionnelles directement composables. Pour réaliser ceci, la spécification des propriétés requises et fournies décrites par les interfaces doit être homogène et non ambigüe. De plus, la relation qui spécifie les conditions sous lesquelles le contrat peut être établi doit être spécifiée.

Les approches décrites dans la littérature sont souvent homogènes au niveau du type d'analyse présenté précédemment : soit elles réalisent toutes les analyses sur la configuration finale, soit elles réalisent les analyses par l'établissement de contrats entre les interfaces de chaque composant.

Au niveau des connecteurs, on remarque également deux approches différentes. La première consiste à utiliser des connecteurs logiques, dans ce cas les interfaces ainsi connectées doivent être homogènes. À l'inverse, dans le cas où les interfaces sont hétérogènes en un ou plusieurs points, un connecteur possédant une "glue" est nécessaire afin d'effectuer la connexion.

Dans tous les cas, comme souligné dans [53], afin d'être capable de réaliser des analyses sur une architecture logicielle, une sémantique formelle doit être utilisée pour chacun des éléments de l'architecture. Puisqu'un style architectural permet de définir des types de composants, de connecteurs et d'interfaces spécifiques, ils peuvent être utilisés pour forcer l'utilisation d'une sémantique particulière. Il est donc important d'avoir une spécification explicite du style architectural utilisé. L'intérêt d'un style architectural est alors double :

1. il permet d'imposer, d'une part une sémantique formelle à chacun des éléments de l'architecture, et d'autre part un ensemble de contraintes de connexion permettant de garder une configuration analysable.
2. il permet également, de part les contraintes de connexion qu'il exprime, permettre l'application de principes de génie logiciel afin de posséder des qualités (au sens du génie logiciel) telles que la réutilisabilité.

On voit ainsi émerger plusieurs point important afin de permettre le développement d'un système embarqué et temps réel en s'appuyant sur une architecture logicielle. Premièrement l'utilisation de modèles paraît primordiale afin de se concentrer sur les concepts plutôt que sur le langage. Deuxièmement, la description et la gestion des propriétés extra-fonctionnelles doit être adaptée aux contraintes à analyser : il est intéressant d'utiliser l'établissement de contrats de QoS afin de créer des composants réutilisables dans plusieurs configurations ; cependant, l'analyse d'une configuration peut être nécessaire à l'établissement d'un contrat pour les propriétés émergentes. Afin de réaliser ces analyses, chaque élément de l'architecture doit posséder une sémantique opérationnelle formelle. Enfin, un style architectural, en donnant des types de composants et de contraintes structurelles permet d'obtenir un système analysable, mais aussi certaines propriétés de génie logiciel telles que la réutilisabilité.

Maintenant que les principes pour la mise en place d'une architecture logicielle ont été étudiés, la partie suivante présente les principes de structuration et les styles architecturaux qui ont permis le développement de systèmes informatiques indépendamment de leur cible de déploiement.

2.3 Les principes de l'indépendance

2.3.1 Introduction

La réutilisation de logiciel est un objectif recherché depuis le début de la programmation. Cette préoccupation a fait l'objet de beaucoup de propositions ces vingt dernières années : librairies réutilisables, méthodes et outils d'ingénierie spécifiques au domaine, réutilisation de motifs, architectures logicielles, programmation orientée composant, etc. [49]. À titre d'exemples, Unix a clairement été pensé dans un but de réutilisation ; le langage C fournit des concepts minimaux pouvant être étendus par des librairies réutilisables ; le langage C++ a également été pensé dans le but d'améliorer la réutilisation [114].

Une des tendances actuelles visant à plus de réutilisation est la séparation, lors du développement, des aspects indépendants de la plateforme de ceux dépendants de la plateforme. L'indépendance vis-à-vis de la plateforme est une qualité d'un modèle qui identifie le degré d'abstraction de ce modèle par rapport aux caractéristiques d'une plateforme technologique particulière [3]. Comme sous-entendu dans cette définition et souligné dans [65], un modèle ne peut donc jamais être totalement indépendant de la plateforme sous-jacente. En effet tout modèle est, au minimum, dépendant de son méta-modèle qui constitue alors une plateforme technologique plus ou moins abstraite.

Cette tendance a été renforcée par la proposition MDA (*Model Driven Architecture*) [95], initiée par l'OMG (*Object Management Group*) [94] en 2001. MDA est une philosophie de développement d'architectures basée sur l'utilisation de modèles.

La notion d'indépendance étant générale, ce chapitre se propose d'étudier comment les notions de réutilisation et d'indépendance vis-à-vis de la plateforme on été considérées dans plusieurs domaines de l'informatique.

Avant de présenter ces travaux, la notion de plateforme est discutée.

2.3.2 Notion de plateforme

La notion de plateforme ne connait pas de définition précise et communément admise. MDA [95] propose la définition suivante :

A platform is a set of subsystems and technologies that provide a coherent set of functionality through interfaces and specified usage patterns, which any application supported by that platform can use without concern for the details of how the functionality provided by the platform is implemented.

Cette définition explique qu'une plateforme est un ensemble de sous systèmes et de technologies fournissant un ensemble cohérent de fonctionnalités au travers d'interfaces et de modèles d'utilisations spécifiés. Ainsi, toutes les applications supportées par cette plateforme peuvent utiliser ces services sans se soucier des détails concernant leur implémentation.

Cette définition reste très générale. Une plateforme est perçue différemment suivant l'utilisation qui en est faite. Ainsi, pour un développeur d'applications largement distribuées, un intergiciel est une plateforme. Contrairement à cela, pour un développeur d'intergiciels c'est le système d'exploitation qui est la plateforme. Enfin, si on développe un système d'exploitation ou un système dédié, le matériel, en termes de processeur, de mémoire, etc., est considéré comme étant la plateforme. La notion de plateforme est donc fortement influencée par le domaine. Elle peut être à rapprocher de la notion de sémantique opérationnelle d'un langage de modélisation. La sémantique opérationnelle d'un langage décrit comment un programme valide est interprété pour être exécuté. De la même manière, chacun des services d'une plateforme pouvant être utilisé par un programme valide permet de décrire comment les appels de ce programme aux services de la plateforme vont être exécutés.

On conserve de cette discussion la nécessité de définir une plateforme comme un ensemble de composants en boîtes noires (ou grises) qui, une fois connectés à une application permet de figer sa sémantique opérationnelle. On remarque que la notion de plateforme et la notion de configuration technique sont proches. Dans la suite de ce document, nous utilisons le terme plateforme plutôt que celui de configuration technique.

La section suivante présente comment, au cours du temps et selon les domaines, l'informatique s'est détachée d'un type de plateforme particulier.

2.3.3 Les approches existantes pour l'indépendance vis-à-vis d'une plateforme

2.3.3.1 L'indépendance vis-à-vis de l'architecture matérielle

Le langage de base compréhensible par une machine, appelé langage machine, est composé d'une suite de nombres binaires. Afin de faciliter l'utilisation des ordinateurs pour les développeurs, le langage assembleur a été proposé. Un tel langage est compréhensible par un humain mais reste proche du langage machine et dépend donc étroitement du type de processeur utilisé. Chaque processeur possède son propre langage machine. Ainsi, un programme développé pour une machine ne peut pas être exécuté sur une autre machine. Afin de contourner ce problème, des langages indépendants des instructions d'un processeur particulier ont vu le jour.

Si ces langages permettent de cibler de la même manière plusieurs processeurs, l'accès aux divers périphériques, comme par exemple un port parallèle ou un disque dur, reste toujours dépendant de la plateforme sous jacente. Cet accès requiert une connaissance exacte du disque et une maîtrise fine de la programmation. De plus il demande une gestion des accès concurrents entre les différentes applications qui utilisent ce périphérique. Afin d'aider le développeur, les architectures matérielles des ordinateurs se sont alors vues dotées d'un système d'exploitation.

Les systèmes d'exploitation ont deux buts principaux : gérer les accès concurrents aux ressources et fournir une abstraction du matériel présent sur la configuration technique sous jacente [118]. Pour cela ils fournissent une API (*Application Programming Interface*) commune à une classe de matériels pouvant réaliser cette API. Ainsi, deux disques durs différents peuvent être utilisés de manière transparente malgré une différence de capacité ou de technologie car ils supportent tous les mêmes services d'accès (primitives *open(), read(), write, close(), ...*). Les adaptations nécessaires entre les primitives de l'API du système d'exploitation et celles du matériel sont alors à la charge du pilote de périphériques (*driver*). Le pilote doit donc avoir une connaissance de l'API désiré afin de réaliser l'adaptation nécessaire, dépendante d'un périphérique particulier [119]. Sur un système d'exploitation moderne, l'ensemble de ces APIs est appelé HAL (*Hardware Abstraction Layer*). Les systèmes d'exploitation permettent ainsi de cibler indifféremment les différents matériels classiques d'un ordinateur personnel. Cette architecture est alors une architecture en couche où chacune des couches abstrait une partie des détails matériels par une API (cf. figure 2.5).

La multiplication des systèmes d'exploitation a fait émerger deux problèmes :

1. L'impossibilité d'exécuter simultanément plusieurs systèmes d'exploitation sur la même configuration technique. Ceci est dû à la nécessité d'avoir une gestion centralisée des accès concurrents à un périphérique.
2. L'impossibilité d'exécuter la même application sur plusieurs systèmes d'exploitation. Ceci est dû à l'hétérogénéité des APIs des systèmes d'exploitation.

La notion de machine virtuelle permet de contourner ces problèmes. Une machine virtuelle est une couche supplémentaire pouvant se placer à différents endroits et permettant de fournir différents types de services suivant le but recherché [103].

Une machine virtuelle peut fournir des services semblables à ceux d'une configuration tech-

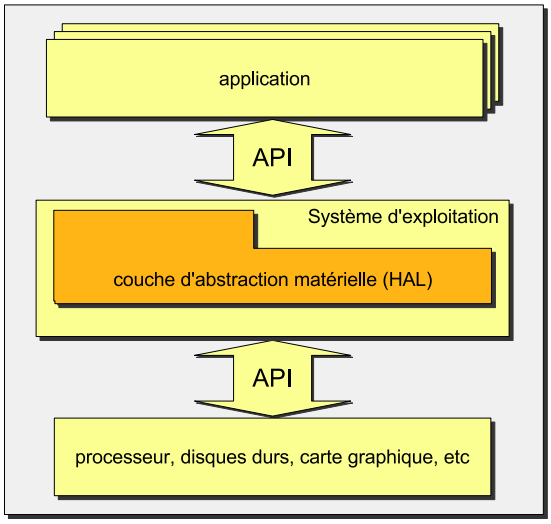


FIG. 2.5 – Les différentes couches d'un système d'exploitation

nique afin de permettre l'exécution de systèmes d'exploitation de manière concurrente (cf. figure 2.6 B). Pour cela, la machine virtuelle, basée sur un système d'exploitation hôte, propose une API semblable à celle d'une configuration technique. Ainsi, les systèmes d'exploitation communiquent avec le matériel virtualisé par l'API de la machine virtuelle plutôt qu'avec le matériel réel ; et ceci de manière transparente.

Une machine virtuelle peut également se placer au dessus d'un système d'exploitation afin d'homogénéiser les APIs de différents systèmes d'exploitation et ainsi permettre à une application de s'exécuter indifféremment sur l'un ou l'autre des systèmes d'exploitation possédant cette machine virtuelle (cf. figure 2.6 A).

Conclusion sur l'indépendance vis-à-vis de l'architecture matérielle Depuis la création des langages adressant plusieurs processeurs jusqu'à l'utilisation d'une machine virtuelle en passant par l'ajout d'une couche HAL, l'envie de donner une interface commune (virtuelle ou non) à tous les matériels est omniprésente. Ainsi, plutôt que de développer un système dédié à un matériel spécifique, l'utilisateur développe un système dédié à une interface donnée, cette interface étant par la suite reliée à des matériels spécifiques.

Les approches suivies s'appuient sur une architecture en couche : chaque couche du système fournit à la couche supérieure une abstraction des services fournis par la couche inférieure. Le développeur base donc le développement de son application sur l'API d'un matériel abstrait. Les services de ce matériel abstrait sont adaptés au fur et à mesure des couches pour correspondre in fine à un matériel réel.

Ce style architectural en couche (*Layered*) a également été adopté dans les réseaux de communication. Le paragraphe suivant détaille ce point.

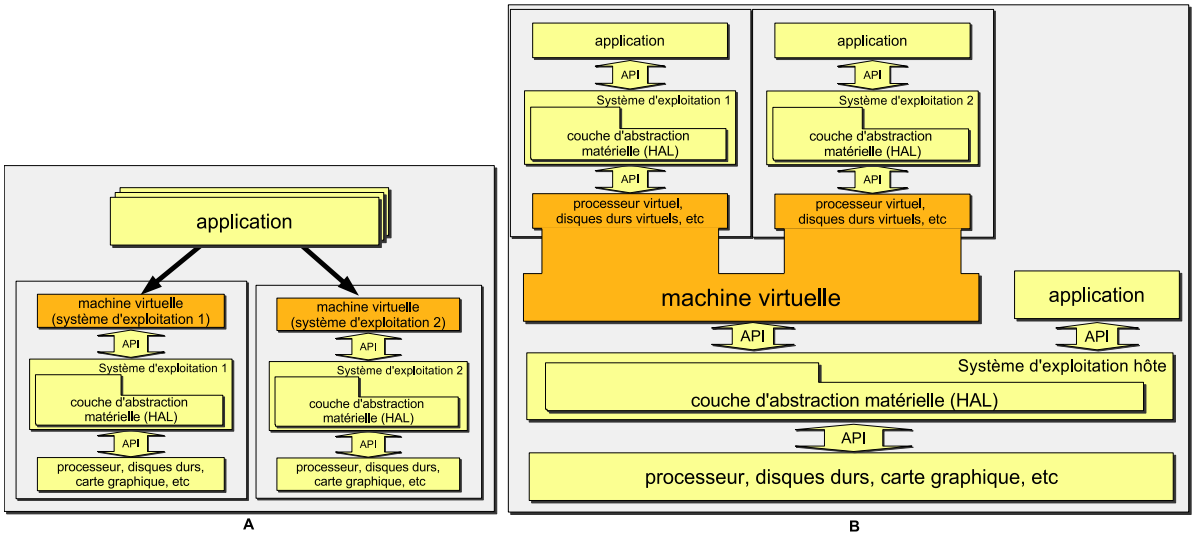


FIG. 2.6 – Les différentes utilisations d'une machine virtuelle

2.3.3.2 L'indépendance dans les réseaux

Dans les années 60, les grands constructeurs (Bull, DEC, IBM, ...) commencent à développer de nombreux protocoles. Malheureusement ceux-ci n'étaient pas compatibles entre eux. Le besoin d'interconnecter tous ces réseaux s'est alors fait de plus en plus pressant, obligeant les constructeurs à se concerter. Parallèlement, la communication entre deux machines sur un réseau a exigé un ensemble d'opérations de plus en plus complexes. De toutes ces contraintes est né le besoin de simplifier la communication en divisant les tâches à effectuer en sous-tâches, chacune implémentée séparément.

C'est en 1982 que commence un effort de standardisation dans le monde des réseaux avec le modèle d'architecture OSI (*Open System Interconnection* [131]). Cet effort fut initié par l'ISO (*International Standard Organization*). Le modèle ISO/OSI est basé sur une structuration en couche. Ce modèle, largement utilisé, est en quelques sortes un modèle général de réseau informatique (cf. figure 2.7). Il répartit les questions relatives au domaine des communications informatiques selon sept couches classées par ordre d'abstraction croissant. Son objectif est d'assurer que les protocoles spécifiques utilisés dans chacune des couches coopèrent pour assurer une communication efficace entre des réseaux et des protocoles hétérogènes. Cependant, cette structuration en couche est couteuse et de ce fait la distance entre l'utilisation concrète (l'implémentation) et le modèle est parfois importante. En effet, peu de programmes peuvent utiliser l'ensemble des 7 couches du modèle : les couches session et présentation sont fort peu utilisées et à l'inverse les couches liaison de données et réseau sont très souvent découpées en sous-couches tant elles sont complexes [24]. De même, pour des raisons de performances, certaines approches telles que [120] court-circuitent l'encapsulation en couches.

Quoi qu'il en soit, les applications développées sur le modèle OSI utilisent des primitives de communications abstraites pour pouvoir transmettre des informations. Ces primitives sont largement plus simples à utiliser que les couches de bas niveau du modèle qui demande une connaissance pointue du réseau physique. L'adaptation entre les primitives offertes aux applica-

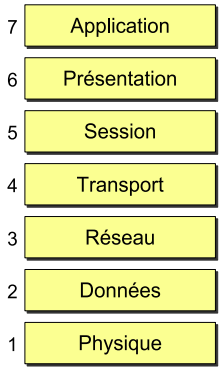


FIG. 2.7 – Les sept couches du modèles OSI

tions et celles qui contrôlent le réseau physique s'effectue au fur et à mesure du passage d'une couche à une autre.

Cependant, avec l'arrivée des systèmes largement distribués, l'utilisation de primitives permettant de masquer les communications distantes entre différents objets s'est fait ressentir. L'utilisation d'un intergiciel (*middleware*) propose alors des primitives semblables pour communiquer entre deux objets distants et deux objets locaux. Il y a de nombreuses définitions différentes du terme intergiciel dans la littérature. Cependant, la définition généralement admise est la suivante :

"*Middleware is the software that assists an application to interact or communicate with other applications, networks, hardware, and/or operating systems*" [16]

Cette définition définit un intergiciel comme un logiciel qui assiste une application pour communiquer avec d'autres applications, réseaux, matériels, et/ou systèmes d'exploitation.

Parallèlement, dans le domaine des réseaux de terrains (FIP [93], Profibus [92], CAN [111], TTP [73]), c'est-à-dire des réseaux dédiés aux communications au sein d'un même système (communication des équipements dans une chaîne d'usine), des besoins différents de ceux des réseaux classiques apparaissent. Du fait du domaine, ces réseaux ont principalement des besoins de prédictibilité et de robustesse. Ils doivent également diminuer les délais de communication des échanges d'informations. Pour ce faire, les réseaux de terrains ont une architecture de communication ne reprenant généralement que trois des sept couches du modèle OSI standard (couche 0, 1 et 7 ; cf. figure 2.7) [100]. Ils laissent ainsi l'adaptation des couches 2 à 6 aux soins du développeur.

Conclusion sur l'indépendance vis-à-vis des réseaux Encore une fois le style architectural en couche permet de s'abstraire des détails matériels. La séparation en sept couches du modèle OSI entraîne des traitements qui peuvent s'avérer coûteux. Pour palier ce coût, certains auteurs comme [23] proposent de court-circuiter cette structure en couches ; réduisant de fait sa modularité. La même préoccupation de performance apparait dans les réseaux de terrains qui limitent le modèle OSI à trois couches fondamentales.

Au contraire, afin de permettre une plus grande transparence dans le développement d'applications distribuées, l'arrivée des intergiciels peut être vue comme l'ajout d'une couche sup-

plémentaire au modèle OSI. Un compromis doit donc être fait entre l'utilisation de nombreuses couches d'abstraction et les besoins de performance.

Si la division en couches des réseaux est normalisée, celle des interfaces homme machine ne l'est pas aussi clairement.

2.3.3.3 L'indépendance vis-à-vis des Interfaces Homme Machine (IHM)

Les premières interfaces hommes machines réalisées étaient spécifiées avec le même langage que le reste du programme. Les interfaces graphiques utilisaient donc les primitives fournies par le système d'exploitation. Alors que les langages informatiques évoluaient et permettaient d'être compilés et/ou interprétés sur différents systèmes d'exploitation, l'utilisation de primitives graphiques ou d'entrée/sortie spécifiques à un système d'exploitation particulier obligeait les applications à être redéveloppées pour chaque nouveau système d'exploitation. L'importance d'une conception architecturale émerge alors et plusieurs styles architecturaux sont proposés.

Le style architectural MVC (*Model-View-Controler*) [74] a été un des premiers à proposer un style architectural réalisant une séparation franche entre la partie présentation et la partie comportement des applications. Il distingue trois composants logiques :
- le `Modèle` réunit l'ensemble des fonctions relatives au noyau fonctionnel de l'application ;
- la `Vue` se charge de la présentation des informations à destination de l'utilisateur ;
- le `Contrôleur` interprète les données en provenance de l'utilisateur.

Ainsi lors d'un changement de plateforme, le `Modèle` peut, en théorie, être réutilisé. Cependant, les communications et les adaptations entre le `Modèle` et les deux autres composants logiques n'est pas explicite et limite donc la réutilisation.

Le style architectural Seeheim [27] distingue également trois composants logiques dans le but de séparer le domaine de l'application de sa présentation à l'utilisateur (cf. figure 2.8). Toutefois contrairement à MVC qui distingue les interfaces d'entrées et celles de sorties, Seeheim les regroupe dans le même composant `Présentation`.

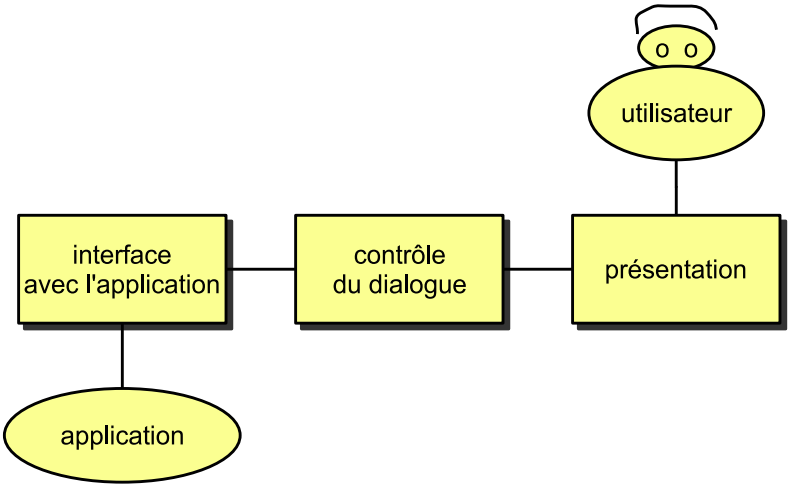


FIG. 2.8 – Le style architectural Seeheim (figure adaptée de [28])

Ce composant lit les données en provenance des dispositifs physiques d'entrée et les traduit en une forme abstraite compatible avec le monde informatique interne. Il reçoit également des informations abstraites qu'il interprète dans les termes du lexique des dispositifs de sorties. C'est le seul composant du système à manipuler les dispositifs physiques d'entrées/sorties.

Seeheim distingue également le composant `Interface avec l'application`. Ce dernier définit les fonctions exportées ainsi que les contraintes d'utilisation de ces fonctions. Enfin, le composant `Contrôleur de dialogue` joue le rôle de médiateur et de filtre entre les deux composants précédemment décrits.

[28] s'accorde avec les nombreux auteurs qui mettent en évidence les avantages de la décomposition en niveaux d'abstractions, depuis les concepts du domaine jusqu'aux fins détails de l'interaction.

Un autre style architectural dit "entrée/sortie" modélise un échange avec l'utilisateur comme une suite d'opérations d'entrées/sorties et de traitements (cf. figure 2.9). Cette suite de transformations est assurée par une hiérarchie de machines abstraites [25, 26]. La machine de plus bas niveau interagit avec les dispositifs matériels servant à communiquer avec l'utilisateur. Les informations en provenance de cette machine sont progressivement transformées en données abstraites compréhensibles par l'application. Dans le sens inverse, les concepts de l'application sont traduits successivement pour être compréhensibles par la machine de plus bas niveau.

Sur la figure 2.9, les flèches verticales symbolisent les appels directs d'un programme client aux fonctions d'une machine. Sur la gauche de la figure les concepts manipulés dans chaque niveau. En caractère gras, on retrouve les classes de services rendus et entre parenthèses, les noms usuels des machines.

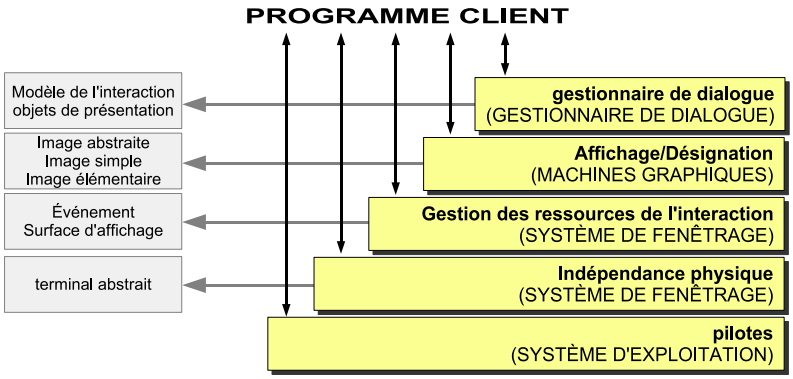


FIG. 2.9 – Le style architectural dit "entrée/sortie" (figure extraite de [28])

Cette décomposition imite le modèle OSI (cf. 2.3.3.2), à la différence près que chacune des couches d'abstraction offre une interface accessible par l'application. Bien que réduisant grandement la modularité, ceci permet de gagner en performance en limitant les coûts de transfert entre les couches logicielles.

Les styles architecturaux "Seeheim" et "entrée/sortie" donnent des principes généraux de structuration et font ressortir l'envie de séparation entre la partie application et la partie présentation. Toutefois, le lien entre ces deux vues est peu détaillé. Le style architectural PAC [29]

pour `Présentation Abstraction Contrôle` comble ce vide. PAC identifie l'interface avec une application dans le composant `Abstraction` et la communication avec le matériel dans la vue `Presentation`. La vue `Contrôle` est alors en charge de l'interaction entre les deux composants précédents. L'interaction est décrite sous forme d'agents [110]. Ce style ne fournit pas de limitation sur l'abstraction idéale entre les composants `abstraction` et `presentation` puisque le composant `contrôle` peut être décomposé récursivement sous la forme d'un système PAC.

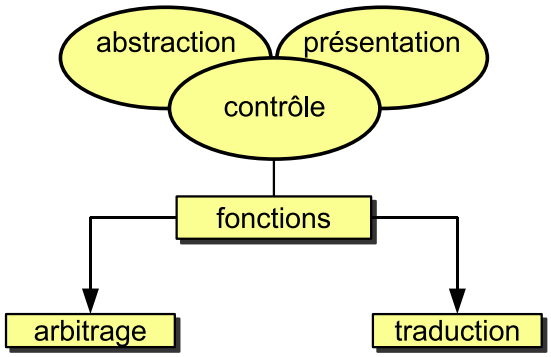


FIG. 2.10 – Le style architectural PAC (figure adaptée de [29])

Le rôle du composant `Contrôle` est identifié. Il est double et comprend les fonctions d'arbitrage et de traduction (cf. figure 2.10). L'arbitrage concerne, par exemple, les synchronisations entre les différents événements en provenance / à destination des composants `Abstraction` et `Présentation`. La fonction de traduction sert, elle, d'intermédiaire entre le formalisme de représentation abstrait de l'application (`Abstraction`) et le formalisme de représentation pour l'utilisateur (`Présentation`).

Conclusion sur l'indépendance vis-à-vis des IHMs Dans le domaine des IHMs, il est courant de constater qu'à une sémantique donnée correspondent plusieurs formes syntaxiques. Autrement dit, l'utilisateur peut déclencher la même fonctionnalité d'une application de différentes manières. Il en est de même pour l'application qui peut communiquer la même information à l'utilisateur de différentes manières. Ceci à amené à proposer des styles architecturaux séparant la partie sémantique et la partie syntaxique. Pour ce faire, on trouve la stratégie de structuration en couche du style "entrée/sortie" et la stratégie de structuration en trois parties de MVC, Seeheim et PAC.

La décomposition en niveaux d'abstraction fournit une base saine à la définition d'architectures modulaires. Cependant, elle introduit des coûts de communication entre chacune des couches qui en limitent les performances. Les styles architecturaux basés sur le modèle Seeheim ont été largement utilisés dans le domaine de l'IHM. PAC raffine et spécifie plus précisément le style Seeheim afin de le rendre plus exploitable. Il est donc important de décrire précisément le rôle de chacun des éléments d'un style architectural ainsi que les principes de réalisation de ces rôles.

Les IHMs sont une famille d'interfaces de communication particulières puisqu'elles utilisent largement l'écran comme moyen d'interaction avec leur environnement (dans ce cas l'utilisateur). Une IHM peut alors évoluer dynamiquement lors de l'interaction (ouverture de nouvelles

boîtes de dialogue, fenêtres, etc.). Dans le domaine qui nous intéresse, les communications avec l'environnement se font par l'intermédiaire de périphériques physiques (capteurs et actionneurs).

2.3.3.4 L'indépendance vis-à-vis des périphériques

Sur un ordinateur classique de type PC, les entrées physiques correspondent essentiellement au clavier et à la souris. De la même manière que la couche d'abstraction logicielle (HAL) d'un système d'exploitation fournit une API indépendante d'un matériel particulier, une API indépendante d'un type précis de clavier ou de souris peut être réalisée. Dans Windows, cette API regroupe les actions possibles d'une souris ou d'un clavier dans un périphérique virtuel [85]. Les applications interagissent ainsi avec une souris virtuelle plutôt qu'avec la souris physique. Ceci permet de lier l'application avec tous les types de souris mais également d'autres types de périphériques rendant les mêmes services qu'une souris (i.e. un *touchpad* ou une palette graphique). Les pilotes des périphériques réels doivent réaliser l'adaptation entre les services du périphérique réel et ceux du périphérique virtuel.

Les problèmes introduits par la communication avec les périphériques d'entrées / sorties d'un système sont rarement traités (cf. analyse de [119]). Cependant, [1] propose une approche à base d'intergiciel pour adresser le problème de l'indépendance d'une application temps réel par rapport à ses capteurs et ses actionneurs [1]. Cette approche a pour but de permettre le développement d'une application indépendamment des types et des unités des données véhiculées par les capteurs ou vers les actionneurs (cf. figure 2.11).

Pour ce faire, le style architectural proposé sépare les données utiles à une application de celles véhiculées au travers des capteurs / actionneurs par une couche de transformation (*Input / Output Transforms* figure 2.11). Puisque les auteurs s'intéressent au développement d'application temps réel, l'étude introduit dans la spécification des données nécessaires à l'application une description qui comprend, pour chacune des données en entrée ou en sortie : l'unité, le type, l'intervalle de valeurs possibles, la validité, la fraîcheur, le taux de rafraîchissement, la précision et la fiabilité.

La couche `transformation` est au cœur de la séparation entre les capteurs / actionneurs et les entrées / sorties nécessaires à l'application. Elle est en charge des conversions de type (par exemple *short integer* vers *integer*, etc.) et d'unité (par exemple km/s vers m/s, etc.). Cette couche est également en charge de la gestion de la redondance. Ainsi, elle peut recevoir les informations de plusieurs capteurs redondants et choisir à l'aide de *voters* l'information qui semble la plus correcte. Dans le sens inverse, elle peut recevoir les informations de plusieurs applications s'exécutant sur des processeurs différents et décider quelle information envoyer vers les actionneurs du système. L'utilisation de *voters* prend en charge certains aspects de tolérances aux fautes. Le développement de l'application basé sur les entrées / sortie est donc indépendant des aspects de tolérances aux fautes.

Malgré la spécification de contraintes temporelles sur les données, aucune information n'est fournie quand à leur utilisation pour la validation du système. Les auteurs ne précisent pas comment sont traitées les propriétés extra-fonctionnelles. De même il n'est pas précisé s'il est possible de construire une entrée à partir de plusieurs informations capteurs différentes.

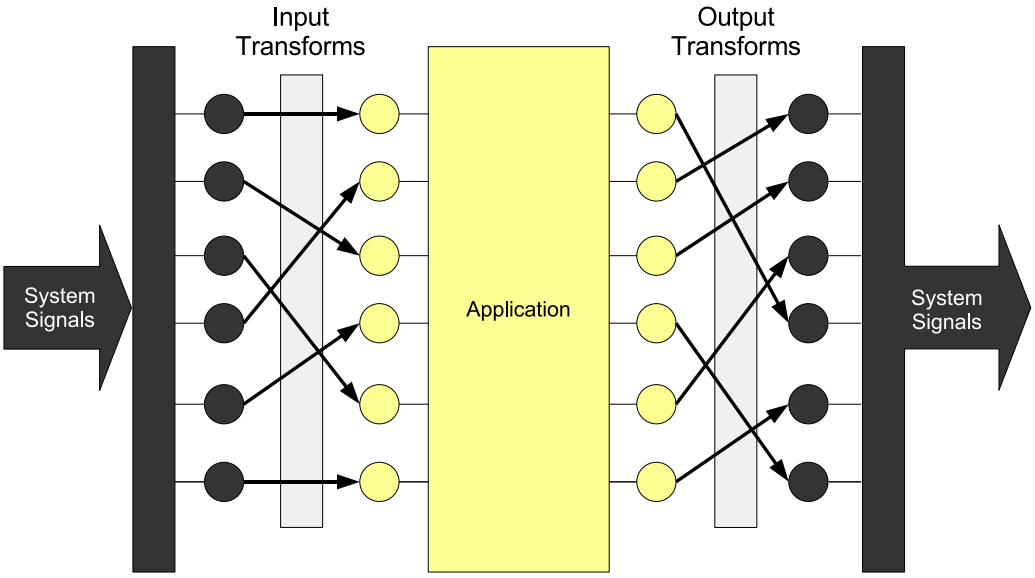


FIG. 2.11 – Un style architectural pour l'indépendance vis-à-vis des capteurs / actionneurs (figure extraite de [1])

Conclusion sur l'indépendance vis-à-vis des périphériques Le fait de virtualiser un composant existant pour baser les applications sur le composant virtuel plutôt que le composant réel rejoint l'idée développée dans [1]. Cependant, en choisissant cette stratégie, les services utiles du périphérique réel sont limités à ceux du périphérique virtuel. Ainsi, on ne peut pas profiter de tous les services du périphérique réel s'ils ne sont pas pris en charge par le périphérique virtuel. Malgré cet inconvénient fort, utiliser un périphérique virtuel permet d'offrir une interface standard entre un ensemble d'applications et un ensemble de périphériques, favorisant ainsi la réutilisation de l'un ou de l'autre.

Dans la même idée que PAC (cf. chapitre 2.3.3.3) pour les IHMs, l'idée décrite dans [1] est intéressante car elle consiste à développer une application en se basant sur une description des entrées et des sorties nécessaires à l'application plutôt que sur une description des données en provenance et à destination des capteurs et des actionneurs d'une plateforme particulière. Cela permet l'acceptation de changement matériel sur les capteurs / actionneurs sans pour autant affecter l'application. Il est toutefois impératif de spécifier de manière précise les entrées / sorties nécessaires à l'application ; d'un point de vue fonctionnel et extra-fonctionnel. Une couche de liaison est alors inévitable à la liaison entre les entrées / sorties et les capteurs / actionneurs. Enfin, il faut s'assurer de la correction fonctionnelle et extra-fonctionnelle des capteurs / actionneurs vis-à-vis de la description des entrées / sorties. Ce point n'est pas abordé dans [1].

2.3.3.5 Conclusion

Les styles architecturaux étudiés dans le cadre de cette thèse montrent indubitablement que le style architectural en couche est le plus adapté pour mettre en œuvre la séparation entre les aspects dépendants et indépendants d'une plateforme.

On distingue deux approches de structuration en couche. La première est celle qui consiste, comme dans la section traitant de l'indépendance vis-à-vis de l'architecture matérielle (section 2.3.3.1), à ajouter successivement des couches afin d'atteindre petit à petit les objectifs d'indépendance désirés. Dans cette approche le nombre de couches peut varier au cours du temps selon les besoins.

L'autre stratégie consiste à élaborer un découpage en couche a priori qui sert de base pour les applications dans un domaine particulier (cf. les couches du modèle OSI section 2.3.3.2). Dans ce cas, plus le nombre de couches est élevé, plus les niveaux d'abstractions sont nombreux. Les aspects dépendant de la plateforme peuvent ainsi s'appuyer sur le niveau d'abstraction qui leur correspond le mieux et éviter la ré-implémentation des couches supérieures pour atteindre la couche la plus haute (la plus abstraite). Le nombre de couches d'abstractions est proportionnel au gain en modularité. Il est cependant inversement proportionnel aux performances. Il est donc nécessaire de réaliser un compromis entre la modularité et les performances.

Pour s'abstraire des périphériques, la virtualisation d'un périphérique existant permet de fournir une interface commune à toutes les applications utilisant cette famille de périphériques. Cette approche n'admet, cependant, que de faibles variabilités entre les fonctionnalités des périphériques réels.

Les IHMs, de leur côté, ont des préoccupations de performances afin d'assurer une interaction efficace avec l'utilisateur. De plus, une IHM, contrairement à un réseau, est réalisée pour interfacer une application particulière. Le nombre de couches minimal permettant l'indépendance est alors recherché. Dans ce contexte, la littérature identifie trois couches : l'interface avec l'application, l'interface avec l'environnement et la couche de liaison entre les deux précédentes. La couche de liaison permet de connecter les deux autres couches en réalisant diverses adaptations suivant les approches. Cette approche est également celle utilisée dans [1] pour obtenir une forme d'indépendance vis-à-vis des capteurs et des actionneurs d'un système.

Cependant, pour les systèmes temps réel, aucune gestion des propriétés extra-fonctionnelles n'est proposée lors de la connexion entre les couches applications et la plateforme sous forme de capteurs et d'actionneurs.

L'apport de MDA Les styles architecturaux décrits précédemment présentent les différentes entités qui s'exécutent dans le système. MDA propose une modélisation de ces entités au travers de trois familles de modèles :

PIM *Platform Independent Model.* Ce modèle permet de spécifier les entités intervenant dans les fonctionnalités d'un système indépendamment de la façon dont ces fonctionnalités seront réalisées par une plateforme.

PM *Platform Model.* Ce modèle permet de spécifier les différentes entités et services fournis par une plate forme particulière.

PSM *Platform Specific Model.* Ce modèle exhibe la façon dont les fonctionnalités des entités décrites dans le PIM sont réalisées par un PM particulier.

Le passage d'un PIM à un PSM se fait par une transformation appelée *mapping* (cf. figure 2.12). Ce *mapping* peut être complexe et ainsi fusionner les deux modèles PIM et PM avec un objectif spécifique, par exemple l'optimisation des performances. Ceci permet ainsi de spécifier une structuration en couche au niveau modèle sans pâtir du coût introduit par une structuration en couche lors de l'exécution.

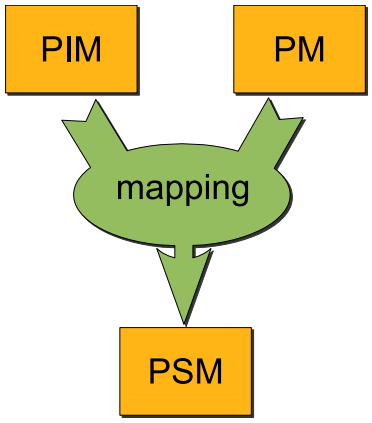


FIG. 2.12 – Les principaux modèles MDA et leurs relations

[3] met en avant le manque de directives dans MDA pour le choix d'abstraction dans le PIM. Il préconise alors l'utilisation d'une plateforme abstraite. Une plateforme abstraite définit un ensemble de plateformes qui sont pertinentes pour l'application. Plus précisément, une plateforme abstraite définit un ensemble de services dont a besoin l'application. La description de ces services peut être plus ou moins précise suivant le degré d'indépendance souhaité. Ceci permet de définir clairement le degré d'indépendance des modèles PIM en réduisant la taille de l'espace de modélisation à explorer pour la réalisation des PSMs.

[3] identifie alors deux approches pour la réalisation des PSMs. Dans la première (cf. (1) figure 2.13), la plateforme réelle est adaptée pour correspondre directement à la plateforme abstraite. Le modèle PIM n'est donc pas modifié. Dans la deuxième approche (cf. (2) figure 2.13), le PSM est une adaptation, en respect de la spécification réalisée dans le PIM, permettant de le composer avec la plateforme réelle.

Dans le domaine des IHMs, on peut citer l'approche SEFAGI [21]. Ce travail permet de décrire une plateforme abstraite ainsi que les adaptations prédéfinies pour un ensemble de plateformes réelles. Ceci permet de changer les adaptations de la plateforme abstraite selon la plateforme réelle à l'exécution. Ainsi l'application s'adapte dynamiquement à la plateforme. Il faut cependant réaliser a priori les adaptations nécessaires et donc définir un ensemble fini de plateformes réelles acceptables.

Décrire explicitement une plateforme abstraite permet de mieux comprendre les caractéristiques présumées des plateformes utilisables et permet ainsi une réutilisation plus aisée d'un PIM sur différentes plateformes. De plus, dans le cadre des systèmes embarqués temps réel, cela permet de donner les caractéristiques extra-fonctionnelles devant être respectées par la plateforme réelle.

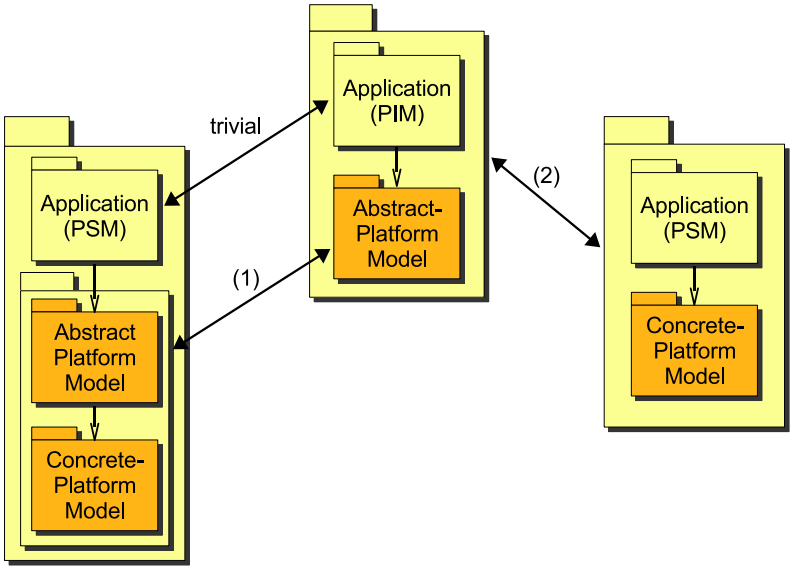


FIG. 2.13 – Deux approches pour la réalisation d'un PSM (figure extraite de [3]).

2.4 Conclusion sur l'état de l'art

Le domaine des architectures logicielles a donné lieu à de nombreuses approches, souvent destinées à une utilisation particulière. Ainsi, certaines approches sont destinées à permettre l'installation ou le remplacement d'éléments de l'architecture (composants) lors de l'exécution ; d'autres permettent de générer un code exécutable à partir d'une description architecturale ; enfin d'autres encore s'attachent à fournir des descriptions architecturales permettant la réalisation d'analyses de performances.

Nonobstant cette diversité entre les approches, des similarités apparaissent selon le domaine adressé. Ainsi, classiquement dans le domaine des systèmes embarqués temps réel, le comportement d'un composant logiciel est réalisé en langage C, puis compilé et exécuté sur un système d'exploitation temps réel. Dans la quasi totalité des approches destinées aux applications temps réel, la description architecturale résulte en une configuration opérationnelle. Travailler au niveau de la configuration opérationnelle permet d'analyser de manière précise un système dédié à une plateforme particulière. Cependant, afin de décrire une architecture de manière indépendante d'une plateforme d'exécution particulière, il est possible de décrire une architecture logique et d'abstraire la plateforme d'exécution en spécifiant des budgets temporels.

Pour les systèmes embarqués temps réel, les approches se sont principalement attachées à permettre la description et l'analyse de propriétés classiques dans le domaine (ordonnancement, contraintes de bout en bout, vivacité, etc.). Peu de styles architecturaux ont été proposés afin de permettre l'application de principes de génie logiciel. En particulier, la séparation des préoccupations entre la configuration logique et la plateforme n'est que peu abordée. Lorsque ce point est abordé, il décrit la configuration technique en termes de support d'exécution et ne se préoccupe pas ou peu des interfaces de communication entre le système et le monde extérieur. Ce point nous parait pourtant crucial pour permettre l'utilisation d'une même configuration logique sur plusieurs plateformes.

Lors de la connexion des interfaces, on trouve parmi les approches existantes l'établissement de contrats de niveau 1, 2 et 3 mais les contrats de QoS n'en sont qu'à leur début. La principale préoccupation des approches existantes est alors de spécifier des contrats plutôt que de les établir. Ces approches proposent généralement de négocier les contrats de QoS lors de l'exécution. Dans les systèmes visés par l'étude, les aspects critiques imposent l'établissement des contrats avant l'exécution. De plus, afin de prendre en compte les propriétés émergentes, il peut être nécessaire de coupler l'établissement d'un contrat de QoS avec la réalisation préalable d'analyses sur une configuration.

Enfin, pour permettre la mise en application de principes de génie logiciel et aussi permettre l'établissement de contrats de QoS de manière formelle, un style architectural adéquat doit être défini. Le style architectural doit d'une part définir des principes de structuration et des contraintes pour utiliser les résultats issus du génie logiciel et d'autre part définir des types de composants et de connecteurs permettant la description et l'évaluation de propriétés extra-fonctionnelles ainsi que l'établissement des contrats associés.

Ainsi, comme le montre la littérature, afin d'augmenter la réutilisabilité d'une plateforme a une autre, un style architectural en couche est préconisé. De plus, ce style doit permettre la définition d'une plateforme abstraite permettant de spécifier de manière explicite les besoins de l'application. Enfin le style architectural doit également permettre la définition d'une plateforme réelle et de la liaison complexe qui lie la plateforme abstraite à la plateforme réelle. Ces descriptions devront permettre la réalisation des analyses et la mise en place des contrats nécessaires pour assurer la correction du système.

En partant de ce constat, l'étude proposée a pour but de fournir les concepts définissant un style architectural pour assurer l'indépendance d'un système de contrôle de processus vis-à-vis de ses besoins de communication avec le monde extérieur. De plus, les concepts du style architectural proposé doivent permettre de valider la correction de l'application de contrôle en fonction des contraintes temporelles de ses communications avec le monde extérieur. La présentation de ces concepts est réalisée dans le chapitre suivant.

3

SAIA

Sommaire

Ce chapitre présente le style architectural proposé dans le cadre de cette thèse. Appelé SAIA pour *Sensors / Actuators Independent Architecture*, il correspond à un ensemble :

- de types de composants,
- de contraintes sur les interactions possibles entre ces composants,
- de modèles formels pour la description des caractéristiques de QoS, associés à des règles d'évaluation de la QoS,
- de règles d'établissement d'un contrat de QoS.

Les types de composants, leur QoS ainsi que les contraintes sur leurs associations sont exprimées à l'aide de différents modèles. Une première partie de ce chapitre permet d'introduire les principes généraux de structuration des composants selon le style architectural SAIA. Une deuxième partie permet de décrire plus précisément les types de composants, de connecteurs et de contraintes induits par le style SAIA. Enfin, une troisième partie se focalise sur les aspects extra-fonctionnels, ici temporels.

3.1 Le style architectural SAIA

Le style architectural SAIA permet le développement d'un système de contrôle de processus de manière indépendante des capteurs et des actionneurs présents sur une plateforme spécifique. La section 2.3 montre que la structuration en couche est une façon classique de structurer un système lorsque l'on désire être indépendant d'une plateforme ; c'est donc la stratégie adoptée dans SAIA.

3.1.1 La structuration en couches

En se basant sur les idées développées par les modèles de structuration dédiés aux IHMs (notamment PAC et Seeheim) et sur le modèle de structuration pour les capteurs et les actionneurs présenté précédemment, SAIA spécifie le système selon trois couches distinctes. On trouve dans une première couche l'application de contrôle et ses interfaces puis, dans une autre couche, la technologie de communication entre le système et le monde extérieur. Enfin, entre ces deux couches, une troisième couche assure la liaison au travers d'adaptations et de contrôles.

1) La première couche définit l'application et ses interfaces en termes d'*entrées* et de *sorties*. Les *entrées* représentent les données et les phénomènes physiques pertinents pour l'application. Les *sorties*, quant à elles, représentent les actions physiques utiles pour le contrôle du processus. Les *entrées* ainsi que les *sorties* sont spécifiées sans fournir de renseignements concernant la technologie grâce à laquelle elles sont acquises ou réalisées. L'ensemble composé des *entrées* et des *sorties* est une spécification de la plateforme abstraite. Elle fournit un ensemble de services d'écritures et de lectures utilisables en boîte noire ou grise et permettant de réaliser les communications de l'application avec le monde extérieur via des interfaces d'entrées et de sorties. Ceci permet de spécifier l'application en se basant sur les services de la plateforme abstraite et donc indépendamment des services fournis par les pilotes des capteurs et des actionneurs. Par analogie

au PIM (*Platform Independent Model*) décrit dans MDA, ces informations sont contenues dans le modèle appelé SAIM (*Sensors / Actuators Independent Model*).

2) La deuxième couche définit les services de communication entre le système et le monde extérieur. Cette communication est réalisée par des éléments de la plateforme matérielle : les capteurs et les actionneurs. Les capteurs fournissent une vue du monde extérieur et les actionneurs réalisent des actions sur le processus. Plus précisément, cette couche définit les services des pilotes de capteurs et des pilotes d'actionneurs, accessibles au travers d'interfaces d'entrées et de sorties. Il s'agit donc de la spécification de la plateforme réelle en termes de capteurs et d'actionneurs. En rapport avec la terminologie MDA qui identifie un modèle de la plateforme (*PM : Platform Model*), SAIA identifie le modèle des capteurs et des actionneurs : le SAM (*Sensors Actuators Model*).

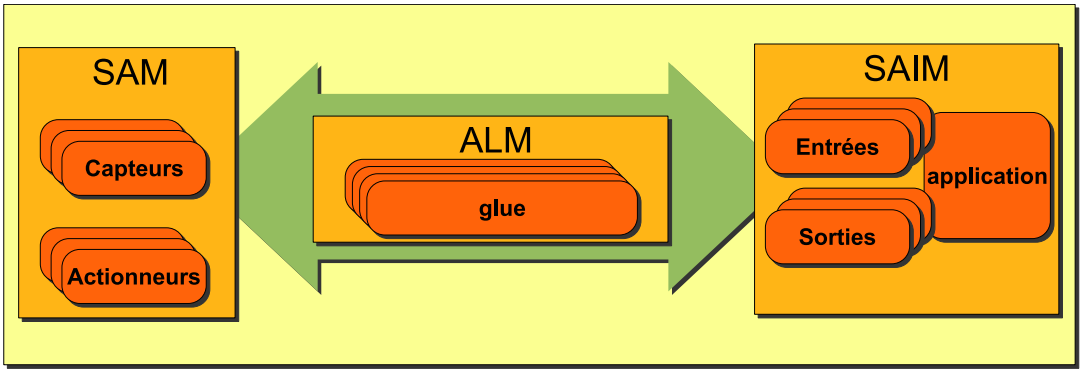


FIG. 3.1 – Les différents modèles dans SAIA

3) La troisième couche identifiée permet aux deux modèles présentés précédemment de spécifier un système complet. Pour ce faire, les opérations d'adaptation, de filtrage et d'arbitrage identifiées dans les composants "contrôle" et "contrôleur de dialogue" de PAC et de Seeheim font partie des opérations nécessaires. Ces opérations sont définies par l'ALM (*Adaptation Layer Model*). D'un point de vue MDA, L'ALM peut être vu comme la spécification du *mapping* qui permet de lier le PIM et le PM. Plus précisément, c'est une adaptation de la plateforme réelle définie par le SAM afin qu'elle corresponde directement à la plateforme abstraite définie par les *entrées* et les *sorties* du SAIM.

3.1.2 Le connecteur complexe

L'ALM est la spécification d'un connecteur complexe qui réalise la liaison entre les interfaces des composants du SAM et celles des composants du SAIM. Les *entrées* et les *sorties* du SAIM sont décrites de manière abstraite alors que les pilotes des capteurs et des actionneurs offrent des services liés aux technologies, les flots d'informations et les interfaces spécifiés par les uns et les autres sont donc, a priori, hétérogènes. Le but du connecteur complexe est de spécifier la "glue" permettant d'homogénéiser les flots d'information entre le SAM et le SAIM.

Le connecteur complexe est composé de sous connecteurs pouvant communiquer les uns avec les autres. Chaque sous connecteur est spécifié par un composant composite dont le type est

spécifique au type d'*entrée* ou de *sortie* auquel il est associé. Puisqu'un sous connecteur est un composant composite, sa "glue" est spécifiée par un assemblage de composants (configuration interne).

SAIA identifie trois types d'hétérogénéités entre les flots d'informations spécifiés dans le SAIM et ceux spécifiés dans le SAM.

La première hétérogénéité concerne les valeurs spécifiées par les flots de données. Celles-ci peuvent être exprimées dans une autre unité, un autre repère, etc. Effectuer les conversions nécessaires à l'homogénéité des valeurs constitue l'étape de formatage.

La deuxième hétérogénéité est due à la différence de niveau d'abstraction des composants du SAM et de ceux du SAIM. Par exemple, le SAIM peut spécifier l'*entrée* `consigne_Vitesse` dans la plateforme abstraite. Les valeurs du flot de données spécifiées par l'*entrée* sont donc des vitesses. Afin de réaliser cette *entrée* sur une plateforme réelle, il est possible d'utiliser un joystick. Les valeurs du flot de données en sortie du pilote du joystick sont des coordonnées. Passer d'une coordonnée à une vitesse (et éventuellement une rotation) constitue une étape d'interprétation nécessaire, programmable, mais non automatisable a priori.

La troisième hétérogénéité concerne la QoS de chacun des flots d'informations. Les tenants et aboutissants de cette source d'hétérogénéité sont décrits plus en détails dans la suite de ce chapitre.

3.1.3 QoS et qualité de contrôle

La séparation en trois modèles distincts permet de réaliser une séparation franche entre d'une part l'application et d'autre part les pilotes des capteurs et des actionneurs d'une plateforme. Cependant, une application de contrôle de processus est sensible à plusieurs paramètres qui influencent la qualité du contrôle. La qualité de contrôle est un ensemble de contraintes exprimées sur le processus en termes de stabilité ou de marges d'erreurs.

Par exemple, pour un robot devant suivre une ligne sur le sol, une des contraintes sur la qualité de contrôle peut être que le robot ne doit pas dévier de la trajectoire désirée de plus de 1cm.

La qualité de contrôle d'une application est fortement influencée par les caractéristiques temporelles concernant l'acquisition des informations en provenance du monde extérieur ainsi que sur la réalisation des actions physiques sur le processus. SAIA propose de dériver les contraintes de qualité de contrôle en contraintes temporelles sur les flots d'informations en provenance et à destination de l'application. Dans le SAIM, cela se traduit par une spécification de la QoS sur la plateforme abstraite qui, lorsqu'elle est satisfaite, assure d'atteindre la qualité de contrôle recherchée par l'application.

Une fois le SAIM et sa QoS spécifiés, il faut être capable de connecter la plateforme abstraite à une plateforme réelle en s'assurant que la QoS est satisfaite. Pour ce faire, il est nécessaire que la plateforme réelle réalise une description de la QoS. La QoS du SAM découle d'une évaluation

de performances des pilotes existants. La QoS dépend de l'architecture des pilotes et du matériel (capteur, actionneur) associés aux pilotes [12, 102]. Les analyses ainsi effectuées permettent d'obtenir une description la QoS pour chacun des pilotes du SAM. Le but de cette étude étant de permettre l'établissement d'un contrat de QoS entre le SAM et le SAIM, les travaux développés dans le cadre de cette thèse présument que la description de la QoS du SAIM et du SAM est disponible.

3.1.4 Le contrat de QoS

Lors de la connexion du SAIM et du SAM, il est nécessaire d'établir un contrat de QoS (contrat de niveau 4) pour garantir la correction de l'application vis-à-vis de sa qualité de contrôle. Ceci se traduit par l'établissement d'un contrat de QoS entre chacune des *entrées* et des *sorties* spécifiées dans le SAIM et chacun des pilotes spécifiés dans le SAM. Ce contrat doit exprimer si la relation de satisfaction entre QoS requise et QoS fournie est vérifiée.

L'établissement d'un contrat de QoS est à la charge du connecteur complexe. Cependant, nous avons vu que le connecteur complexe est composé de sous connecteurs possédant chacun une "glue" afin de rendre homogènes les flots d'informations entre le SAM et le SAIM. En modifiant les flots d'informations, la "glue" modifie également leur QoS, spécifiée dans le SAM et dans le SAIM. Avant de pouvoir établir le contrat de QoS, l'impact de la "glue" sur la QoS est évalué par la réalisation d'analyses temporelles.

Pour assurer la faisabilité de ces analyses, les composants doivent posséder une sémantique opérationnelle formelle ainsi qu'une spécification des informations temporelles pertinentes. Ces informations temporelles reflètent les durées d'exécution des traitements réalisés ainsi que les temps de blocages induits par l'ordonnancement des actions.

La description de la sémantique opérationnelle et des différents temps de traitements permet d'effectuer les analyses temporelles et donc d'évaluer la QoS des flots d'informations modifiés par la "glue" des sous connecteurs. Il est alors possible de vérifier si la connexion entre le SAM et le SAIM donne lieu à l'établissement d'un contrat ou non (cf. figure 3.2).

3.1.5 Niveau de description architecturale

Il est important de noter que l'utilisation du style architectural SAIA amène à la construction d'une architecture logique. En effet, une fois le système décrit avec SAIA, seule la plateforme de communication est définie. La plateforme d'exécution a été abstraite par des informations temporelles sur les durées d'exécution de ses traitements ainsi que par les temps de blocages possiblement induits par l'ordonnancement. Ces informations temporelles sont donc des contraintes devant être respectées lors de la mise en œuvre de l'architecture : on parle de budgets temporels.

Décrire une architecture logique permet d'effectuer une première analyse du système avant son implémentation et son déploiement sur une plateforme d'exécution particulière. Ceci permet d'effectuer une première validation d'un système de contrôle lors de la connexion de la plateforme

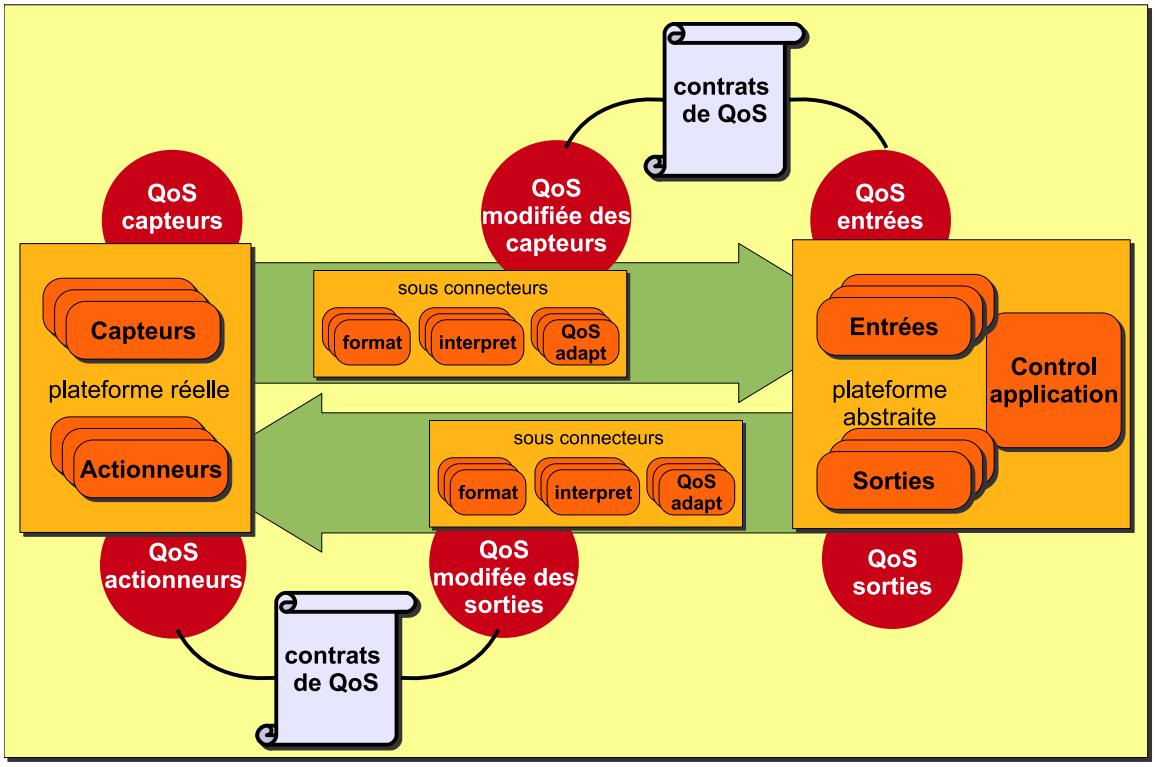


FIG. 3.2 – Gestion de la QoS dans SAIA

abstraite avec la plateforme réelle. Cette première validation intervient avant le choix d'une plateforme d'exécution. Elle est nécessaire pour éviter d'obtenir, sur une plateforme spécifique, une structuration logique fournissant des comportements temporels non cohérents avec la QoS. On évite ainsi les allers-retours coûteux entre la structuration logique et son implémentation.

Raisonner sur une architecture logique est intéressant cependant, lorsque la plateforme d'exécution est connue, les différentes informations temporelles mesurées, telles que les durées d'exécution ou les temps de blocage, peuvent être renseignées dans SAIA sans remettre en cause les possibilités d'analyses temporelles. Toutefois, il faut noter que SAIA ne spécifie ni les types de composants nécessaires à la description de la plateforme d'exécution, ni les éléments nécessaires à la spécification d'un modèle de tâches.

3.1.6 SAIA et le langage

L'expression d'un style architectural nécessite de faire le choix d'un langage de description. Les avantages liés à l'utilisation de (méta-)modèles afin de manipuler des concepts nous permettent de nous affranchir d'un langage de programmation particulier. Ainsi, SAIA décrit le style architectural au travers de différents méta-modèles. Pour des facilités d'outillages les méta-modèles sont exprimés dans une extension d'UML fournie par l'outil GME [62] ; et de contraintes exprimées en OCL [96].

Sur ces bases, SAIA décrit des types de composants associés à différents modèles. Après

avoir présenté le modèle à composant utilisé, les différents types de composants et les contraintes sur leurs associations sont détaillés dans les parties suivantes. Après la description des types de composants, une description de la QoS, de son évaluation dans le connecteur complexe et de l'établissement du contrat de QoS sont fournies.

3.2 Les (méta-)modèles dans SAIA

3.2.1 Le modèle à composant

Au delà d'un style architectural et des principes de structuration qu'il propose, il est nécessaire de choisir un modèle à composant sur lequel appliquer le style. Le modèle à composant retenu ne doit ni comporter de règles de structuration implicite ni posséder une sémantique opérationnelle non désirée. De plus, il est nécessaire de posséder un modèle à composant comprenant les concepts de bases minimaux que l'on retrouve dans la majorité des modèles à composant pour les systèmes visés par l'étude :

- des composants primaires,
- des composants composites,
- des interfaces,
- des connecteurs logiques.

Pour ces raisons nous avons basé le style architectural sur un modèle ad-hoc, basé sur UML et réalisé spécialement pour SAIA.

Dans le domaine visé, le modèle à composant doit de plus posséder une sémantique opérationnelle temporisée formelle. Un composant dans SAIA possède un ou plusieurs service(s) décrits par un automate temporisé IF (Intermediate Format : [17]). En effet, parmi les langages existants, IF offre l'avantage d'être formel, de permettre la programmation concurrente, d'être temporisé et également d'être outillé. De plus, l'exécution d'un automate IF permet de réaliser une simulation exhaustive du comportement de l'automate dont le résultat est accessible sous forme de LTS (*labeled Transition System*). Dans le modèle à composant de SAIA, chaque automate IF peut faire appel à une fonction dont l'exécution est caractérisée temporellement par un budget temporel. Ce budget temporel est lui-même caractérisé par un intervalle précisant le temps minimum et le temps maximum imparti pour l'exécution de la fonction.

Puisque la communication entre l'application de contrôle et le monde extérieur est caractérisée par des flots d'informations, chaque composant comporte des interfaces d'entrées et/ou des interfaces de sorties. Les interfaces d'entrées consomment un flot d'informations alors qu'une interface de sortie le produit. Plus précisément, les interfaces d'entrées permettent à un flot d'informations d'entrer dans un composant pour y être traité. Ceci se traduit par l'appel à un service fourni par le composant en question. L'appel au service est donc fait à chaque nouvelle occurrence dans le flot d'entrée. À l'inverse, les interfaces de sortie permettent à un flot d'informations de sortir d'un composant. Ceci se fait via l'appel à un service requis, défini par un autre composant. Encore une fois, l'appel se fait à chaque nouvelle occurrence du flot de sortie.

Afin de caractériser les occurrences, SAIA définit deux types abstraits d'occurrences : `T_eventtype`, qui caractérise une occurrence d'un flot d'événements, et `T_datatype`, qui caractérise

une occurrence d'un flot de données. Ils contiennent respectivement deux et trois champs :

- *timestamp*, qui représente la date du système à laquelle l'occurrence est créée.
- *initial_ delay*, qui représente le retard de l'occurrence lors de sa création. Si l'occurrence est créée par le système, le retard initial est de 0. A contrario, si l'occurrence est issue d'un pilote d'acquisition, ce retard est le reflet du temps passé entre la lecture de l'information par le capteur physique et la création de l'occurrence correspondante par le pilote.

Puisque `T_datatype` contient des données, il contient également la valeur de l'occurrence, notée *occ_ value*.

Dans le modèle à composant réalisé, à une interface correspond un et un seul service. Ainsi, la liaison entre une interface de sortie et une interface d'entrée est possible uniquement lorsque le service requis par l'interface de sortie correspond au service fourni par l'interface d'entrée. De plus, puisqu'à une interface ne correspond qu'un et un seul service, alors à une interface ne correspond qu'un et un seul flot d'informations. Ceci permet de lier la description de la QoS d'un flot à une interface.

La sémantique opérationnelle des services, des connecteurs et des interfaces dans SAIA est fournie par la sémantique opérationnelle du langage IF. Ainsi, pour les services d'un composant, l'exécution des automates possède la sémantique opérationnelle des automates IF et l'exécution des fonctions possède la sémantique d'un temps d'attente dont la durée est spécifiée par le budget temporel associé à la fonction. La communication entre deux composants dont les interfaces sont reliées par un connecteur logique correspond à une transmission d'occurrence et possède donc la même sémantique que l'envoi d'un message entre deux automates IF. La communication dans SAIA est donc de type asynchrone (émission sans attente).

La communication d'une occurrence est à l'initiative du composant producteur via l'appel à un service du composant consommateur. La communication dans SAIA est donc de type *Push*. Ce choix est classique lorsque l'on traite des flots d'informations. De plus, il facilite la modélisation d'un système SAIA en IF. Cependant passer à une communication de type *Pull*, c'est-à-dire où le traitement d'une occurrence d'un flot est à l'initiative du composant réalisant le traitement reste envisageable et ne remet pas en cause les possibilités d'analyses et/ou d'établissement de contrat de QoS exprimés par la suite.

Enfin, lors d'un envoi de message en IF, celui-ci est déposé dans une boîte aux lettres de type FIFO (*First In First Out*) en attendant d'être consommé. Les interfaces de SAIA possèdent donc la sémantique d'une boîte aux lettres de type FIFO, cependant, nous fixons la taille de la FIFO à 1. Lorsqu'une information est déposée dans une boîte aux lettres non vide, l'information précédente est écrasée par la nouvelle information arrivée. Ce choix permet de s'affranchir des problèmes d'explosion des boîtes aux lettres dans le système. De plus, puisque l'on considère des flots d'informations à destination et en provenance du monde extérieur pour contrôler un processus physique, ne considérer que la dernière occurrence d'une information semble logique pour être réactif plutôt que stocker ces informations dans une FIFO pour un traitement ultérieur. Toutefois, il complique la modélisation d'un système SAIA en IF car la taille de la boîte aux lettres d'un système IF ne peut pas, dans les outils existants, être choisie. On s'assure donc que les boîtes aux lettres peuvent être lues dans tous les états du système afin de mettre à jour une variable interne, représentative de l'information reçue.

Le modèle à composant utilisé dans SAIA est basé sur le paradigme de méta modélisation de l'outil GME. Ce paradigme définit les entités de bases décrites précédemment : les composants primaires (appelés *Atom* dans l'outil), les composants composites (*Model*), les interfaces (*Port*) et les connecteurs logiques (*Connection*). De plus, aucune sémantique opérationnelle n'est associée à ces entités dans GME. Ceci nous a permis de définir un modèle à composant simple et ad-hoc dont les concepts et la sémantique opérationnelle ont été définis spécifiquement dans SAIA. Cependant, tous les modèles à composant qui peuvent correspondre aux critères énoncés précédemment peuvent, de fait, appliquer le style architectural décrit dans cette thèse.

Maintenant que les éléments principaux du modèle à composants sont présentés et que leur sémantique opérationnelle est précisée, les parties suivantes s'attachent à la description des différents types de composants et de contraintes définis par le style architectural proprement dit.

3.2.2 Le modèle SAM

Les types de composants du SAM représentent les différents types de pilotes des capteurs et de pilotes des actionneurs. Ils décrivent la vue d'une plateforme réelle en termes de communication avec le monde extérieur.

Nous appelons "pilote d'acquisition" un composant qui produit un flot d'informations à destination de l'application en fonction des fluctuations d'un ou de plusieurs capteur(s) via l'appel à une fonction spécifique du pilote. Parallèlement, nous appelons "pilote de réalisation" un composant qui reçoit un flot d'informations en provenance de l'application pour faire fluctuer les actionneurs en conséquence (toujours via l'appel à une fonction spécifique du pilote). Les pilotes ne possèdent qu'une et une seule interface : les pilotes d'acquisition une interface de sortie, et les pilotes de réalisation une interface d'entrée. Dans cette étude, nous ne nous intéressons qu'au fonctionnement nominal des pilotes en supposant leur initialisation correcte. Nous différencions également un composant selon si le flot d'informations qu'il manipule est un flot de données ou un flot d'événements.

On distingue donc au final quatre types de composants dans le SAM, chacun de ces types de composants est dédié à l'acquisition ou à la réalisation d'un flot de données ou d'événements :

- les pilotes d'acquisition de données (*driver_ data*),
- les pilotes d'acquisition d'événements (*driver_ event*),
- les pilotes de réalisation de commandes (*driver_ cmd*),
- les pilotes de réalisation de commande événementielles (*driver_ cmd_ event*).

Enfin, chacune des interfaces des composants du SAM est associée à une description de la QoS qui lui est propre. Ces informations sont représentées d'une manière simplifiée sur le méta modèle de la figure 3.3. Sur cette figure, et sur les suivantes, $\ll$ *Model* $\gg$ représente un composant composite. Au contraire $\ll$ *Atom* $\gg$ représente un composant primaire, pouvant contenir des attributs mais ne pouvant pas être composé d'autres composants. $\ll$ *Connection* $\gg$ représente un type de connecteur logique. Chacune de ces entités peut être décrite plus en détail dans un autre modèle. Elle est alors référencée et estampillée $\ll$ `xxx`*Proxy* $\gg$ où xxx représente le type d'entité référencée. Enfin, les attributs des composants peuvent être de type *field*, auquel cas cela représente un type abstrait quelconque, et de type *enum*, auquel cas l'attribut peut prendre une des valeurs prédéfinies dans le métamodèle.

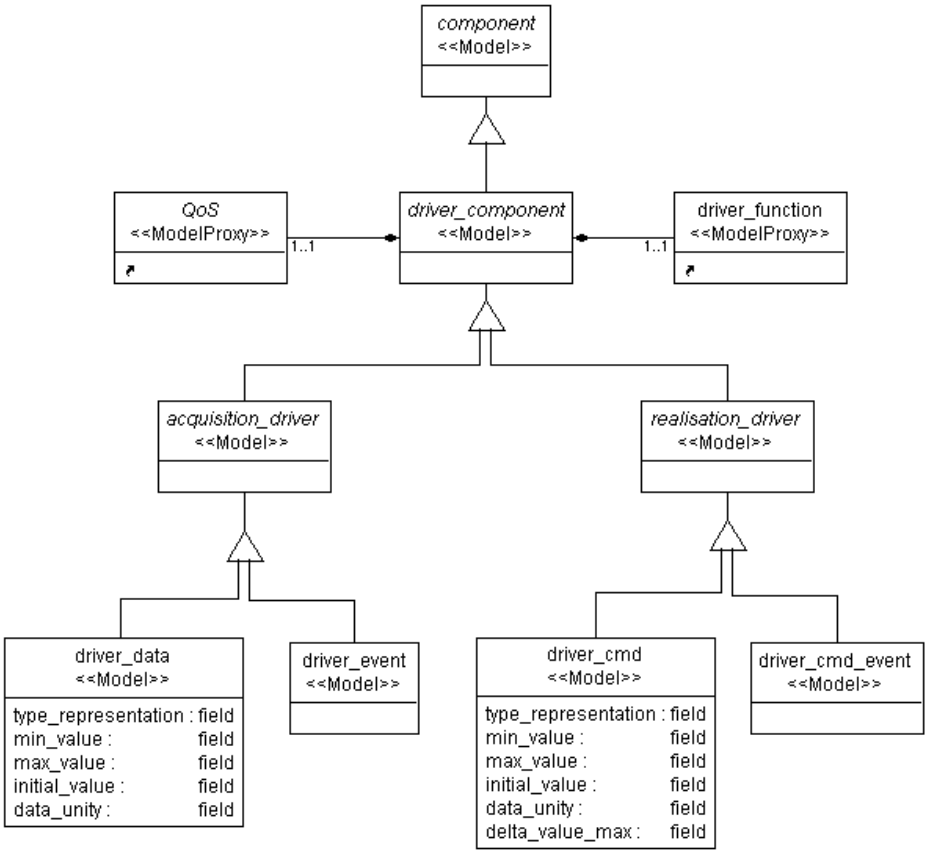


FIG. 3.3 – Métamodèle simplifié représentant les différents types de pilotes

3.2.2.1 Les pilotes d'acquisition de données

Notés `driver_data`, les pilotes d'acquisition de données fournissent une vue de l'évolution d'une donnée physique au cours du temps. Pour cela, ils transforment les informations en provenance d'un ou plusieurs capteur(s) en un flot de données à destination de l'application. Le pilote cache les détails d'accès bas niveau au matériel (registre, brochage, etc.). Le flot de données généré par le pilote d'acquisition de données peut indifféremment être représentatif d'une *Mesure* en provenance de l'environnement ou du processus, où être représentatif d'une *Consigne* en provenance d'un opérateur. Un pilote d'acquisition de données est caractérisé par le flot de données qu'il génère, lui même caractérisé par :

- son nom, représenté par une chaîne informelle de caractères,
- son unité, c'est-à-dire la grandeur choisie comme référence pour mesurer la valeur des occurrences (*data_ unity*),
- son type, c'est-à-dire la représentation informatique de chacune des occurrences (*integer, float, struct, etc.*) (*type_ representation*),
- ses valeurs minimum et maximum, c'est-à-dire les valeurs bornant chacune des occurrences (*min_ value max_ value*),
- sa valeur initiale, c'est-à-dire la valeur de la première occurrence dans le flot (*initial_ value*),
- sa QoS, la QoS est présentée en détail dans le chapitre 3.3.

Le pilote est également composé d'une entité *T_ datatype* permettant de stocker l'occurrence courante du flot de données. Cette occurrence est véhiculée au travers d'une interface *o_ NewDriverData*. Cette interface est spécifiée temporellement par une entité de type *QoS_ data* qui lui est associée via un connecteur logique de type *specifies*.

Afin de permettre au flot de données généré d'être consommé, les composants de ce type requièrent le service :

– `NewDriverData(T_datatype)` //mise à jour d'une donnée en provenance d'un pilote

À titre d'illustration, le métamodèle simplifié d'un pilote d'acquisition de données est donné figure 3.4. Les métamodèles des autres pilotes sont construits de la même manière que celui-ci et, de ce fait, ne sont pas présentés.

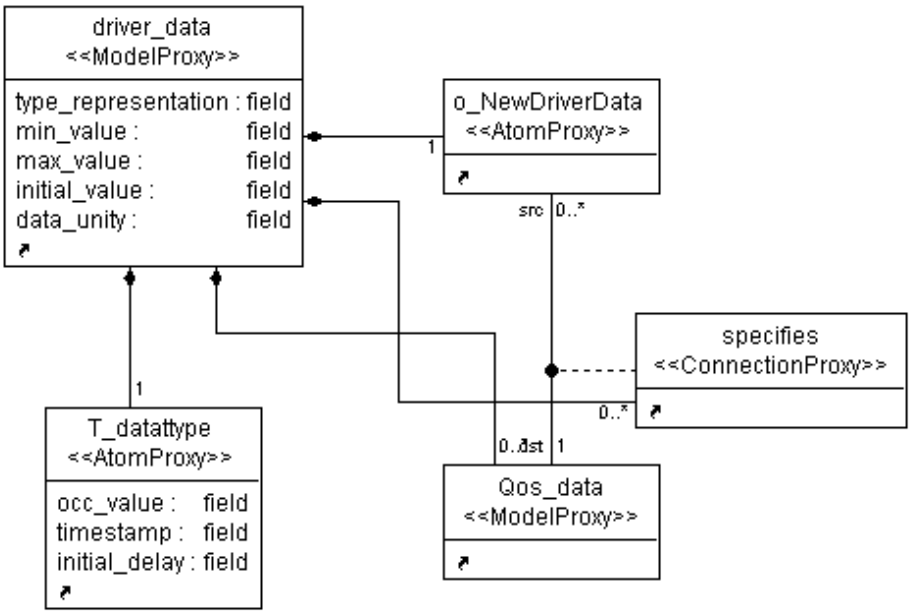


FIG. 3.4 – Métamodèle simplifié d'un pilote d'acquisition de données

3.2.2.2 Les pilotes d'acquisition d'événements

Notés `driver_event`, les pilotes d'acquisition d'événements sont semblables aux pilotes d'acquisition de données au détail près qu'ils fournissent une vue de l'évolution du changement d'un état physique au cours du temps plutôt que d'une valeur physique. De ce fait, aucune valeur n'est définie sur le flot d'événements qui est uniquement caractérisé par :

– son nom, représenté par une chaîne de caractères informelle,
– sa QoS, la QoS est présentée en détail dans le chapitre 3.3.

Afin de permettre au flot d'événement généré d'être consommé, les composants de ce type requièrent le service :

– `NewDriverEvent(T_eventtype)` //arrivée d'un nouvel événement en provenance d'un pilote

3.2.2.3 Les pilotes de réalisation de commandes

Notés `driver_cmd`, les pilotes de réalisation de commandes réalisent des actions pouvant être paramétrées par une valeur dans le but de contrôler le processus. Pour cela, ils consomment un flot de données dont chaque occurrence est une *Commande* en provenance du système. Ils transforment alors ce flot de données en action sur le processus. Encore une fois le pilote est utilisé pour cacher les détails d'accès bas niveau au matériel (registre, brochage, etc.).

Le flot de données, représentatif des *Commandes* du système sur le processus est caractérisé de la même manière que le flot de donnée d'un pilote d'acquisition de données. Contrairement aux pilotes d'acquisition qui possèdent une interface de sortie, les pilotes de réalisation de commandes possèdent une interface d'entrée. À cette interface d'entrée est liée une description de la QoS du flot consommé. Chaque occurrence de ce flot est consommée via l'appel au service fourni :

– `NewDriverCmd(T_datatype)` // mise à jour d'une commande dans la plateforme réelle.

3.2.2.4 Les pilotes de réalisation de commandes événementielles

Notés (`driver_cmd_event`), les pilotes de réalisation de commandes événementielles réalisent une action mécanique non paramétrable dans le but de contrôler le processus. Ils possèdent donc les mêmes caractéristiques que les pilotes de réalisation de commandes mis à part qu'ils consomment un flot d'événements dont chaque occurrence est une *Commande événementielle* en provenance du système. De ce fait, le flot d'événements représentant les *Commandes événementielles* est caractérisé de la même manière que les flots d'évènements des pilotes d'acquisitions d'événements.

Pour consommer le flot d'événements, les composants de ce type fournissent le service :

– `NewDriverCmdEvent(T_eventtype)` // arrivée d'un nouvelle commande événementielle dans la plateforme réelle.

3.2.2.5 Le SAM et la qualité de service

Nous avons vu dans le chapitre contexte (chapitre 2.1) que la QoS des *Mesures*, des *Consignes* et des *Commandes* influe sur la stabilité et la qualité de contrôle du système. Ainsi chaque type de composant du SAM doit décrire la QoS qui caractérise son flot d'informations en termes de loi d'arrivée et de loi de retard sur l'acquisition ou la réalisation des occurrences du flot.

Chacune des caractéristiques de QoS spécifiées par un composant peut être soit du type requise, soit du type fournie. Ce paragraphe discute du type des caractéristiques de QoS dans le SAM.

Les pilotes d'acquisition produisent un flot d'informations, ils génèrent donc les occurrences du flot. De ce fait, ils sont responsables de la loi d'arrivée du flot qu'ils produisent : la loi d'arrivée est donc une caractéristique de QoS fournie. De même, le temps écoulé entre le moment où l'information est vraie dans le monde extérieur et le moment de sa présence en sortie des pilotes d'acquisition est fonction du capteur et de l'architecture des pilotes. La loi de retard est donc également une caractéristique de QoS fournie par les pilotes d'acquisition.

Les pilotes de réalisation consomment un flot d'informations pour réaliser des actions sur le processus. Un pilote de réalisation possède des restrictions sur le flot qu'il consomme. La loi d'arrivée fait partie de ces restrictions car le pilote de réalisation possède un temps minimum nécessaire à la réalisation d'une action. Il est parfois nécessaire de ne pas envoyer une nouvelle commande avant la fin de la réalisation de la précédente afin de ne pas détériorer l'actionneur. Un pilote de réalisation doit donc spécifier la loi d'arrivée requise pour un fonctionnement correct.

Parallèlement à cela, le temps dans lequel l'action va être réalisée par le pilote de réalisation est une information caractérisant le pilote de réalisation. Cette information, puisqu'elle est dépendante du pilote, est une spécification de la loi de retard fournie par le pilote de réalisation.

À part la loi d'arrivée des pilotes de réalisation qui est une caractéristique de QoS requise, les caractéristiques de QoS des types de composants du SAM sont de la QoS fournie. La description formelle de la QoS et de ces caractéristiques est donnée plus en détails dans le chapitre 3.3.

3.2.2.6 Conclusion sur le SAM

Le SAM est un modèle de la plateforme réelle. À ce titre il modélise les services de la plateforme accessibles en boîte grise via ses interfaces. Il est ainsi possible d'utiliser les services des pilotes sans connaissance des mécanismes internes de ceux-ci.

Puisque les services de la plateforme sont utilisés sans connaissance de la manière dont ils sont réalisés, la plateforme peut être réelle ou au contraire peut représenter les services de communication fournis par un simulateur. De cette manière, une cible réelle et une cible simulée sont modélisées à l'identique et sont accessibles via les mêmes services.

Quelque soit la cible représentée, une description de la QoS fournie et requise par les types de composants du SAM est nécessaire pour l'établissement d'un contrat de QoS entre la plateforme réelle du SAM et la plateforme abstraite contenue dans le SAIM dont la description est l'objet de la section suivante.

3.2.3 Le modèle SAIM

Le modèle SAIM est composé de types de composants représentants d'un côté l'application de contrôle et de l'autre la plateforme abstraite caractérisant les besoins de communication entre l'application de contrôle et le monde extérieur.

L'application de contrôle représente le noyau métier du système. C'est elle qui est en charge du calcul des *Commandes* nécessaires pour contrôler le processus en fonction des *Mesures* et des *Consignes* qu'elle reçoit. Aucune attention particulière n'est portée sur l'application de contrôle dans cette thèse. Elle est simplement représentée par un composant accédant à la plateforme abstraite par l'intermédiaire des services suivants :

Services fournis par l'application de contrôle
- `SetSoftData(T_datatype)` //mise à jour d'une donnée dans l'application
- `NewSoftEvent(T_eventtype)` //arrivée d'un nouvel événement dans l'application

Services requis par l'application de contrôle
- `SetCmd(T_datatype)` //mise à jour d'une commande par l'application
- `NewEventCmd(T_eventtype)` //production d'une nouvelle commande événementielle par l'application.

En effet, le but étant de séparer les besoins de communication de l'application (la plateforme abstraite) avec les moyens réels de communication avec le monde extérieur fourni par une plateforme réelle (spécifiés par les pilotes), on se concentre, dans le SAIM, sur la définition de la plateforme abstraite.

La plateforme abstraite est définie par un ensemble d'*entrées* et de *sorties*. Les *entrées* spécifient l'ensemble des *Mesures* et des *Consignes* utiles à l'application pour effectuer le contrôle. Les *sorties*, quant à elles, spécifient les *Commandes* générées par l'application afin de modifier l'état du processus. Chacune des *entrées* et des *sorties* est également différenciée selon qu'elle spécifie un flot de données ou d'événements. On distingue donc quatre types de composant différents dans la plateforme abstraite : les données (*data*), les événements (*event*), les commandes (*cmd*) et les commandes événementielles (*event_ cmd*). Ces quatre types de composant sont représentés sur le métamodèle simplifié de la figure 3.5.

Les deux parties suivantes décrivent la plateforme abstraite et les types de composants intervenant dans sa spécification. Pour ce faire, nous commençons par la descriptions des *entrées* avant de poursuivre avec les *sorties*.

3.2.3.1 Les entrées

Une *entrée* est une vue abstraite du monde extérieur. Le mot "abstrait" reflète le fait que la spécification d'une *entrée* revient à spécifier directement l'élément physique qui est d'intérêt pour l'application. Par exemple, une application de contrôle d'un robot explorateur a de fortes chances de posséder l'entrée '`Position`'. '`Position`' est une valeur physique du processus. Elle représente un besoin de l'application quant à sa vue du monde extérieur. Cependant, elle ne spécifie pas la technologie qui est utilisée pour l'acquisition de la valeur physique (GPS, encodeur de roue, etc.).

Il est important de ne spécifier aucun aspect technologique lors de la description des *entrées* afin de ne pas rendre la plateforme abstraite dépendante d'une technologie particulière.

Pour caractériser les *entrées*, comme pour les pilotes du SAM, on différencie les éléments physiques caractérisés par un flot de données de ceux caractérisés par un flot d'événements. Ils

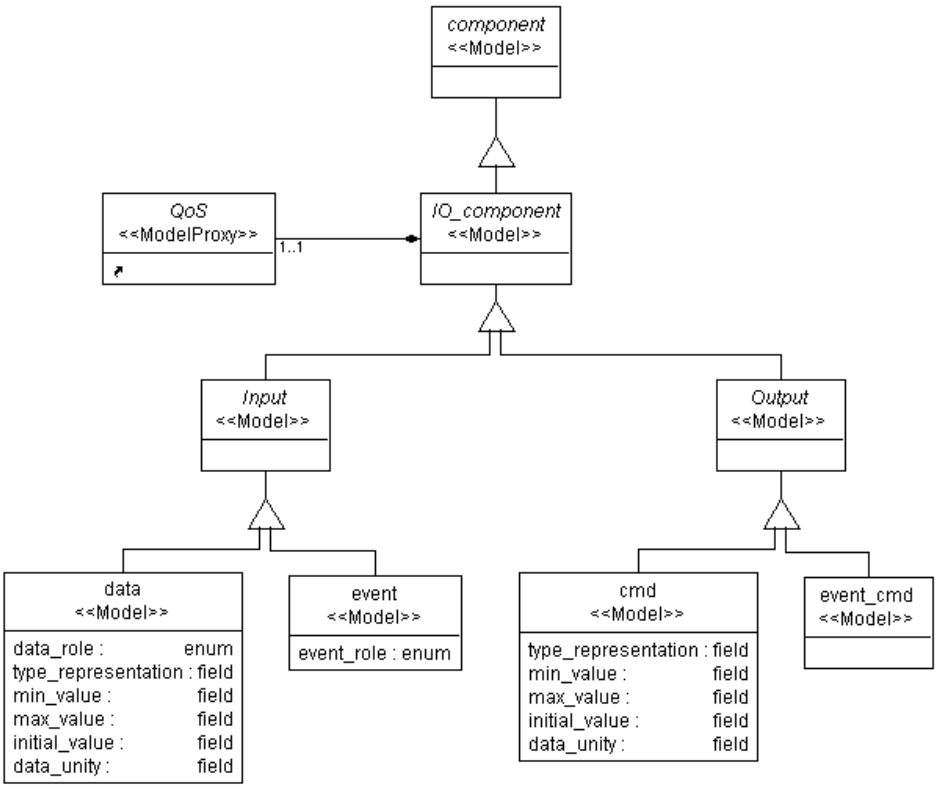


FIG. 3.5 – Métamodèle simplifié représentant les différents types de composant de la plateforme abstraite

sont respectivement nommés '*donnée*' et '*événement*'. Chaque *entrée* est en charge d'un et d'un seul flot d'informations. Une *entrée* ne possède donc qu'une interface d'entrée et qu'une interface de sortie.

On réalise également une autre distinction entre les *entrées* vis-à-vis de leur rôle (*data_role* et *event_role*, figure 3.5). Une *entrée*, qu'elle soit du type *donnée* ou *événement* peut être soit une *Mesure*, soit une *Consigne*. Cette distinction permet de différencier les *Mesures* qui représentent l'état actuel de l'environnement ou du processus, et les *Consignes* qui représentent l'état du processus désiré par l'opérateur. Cette distinction est faite uniquement dans le SAIM car elle reflète une différence applicative. Cette différence n'intervient pas dans le SAM car les *Consignes* et les *Mesures* peuvent provenir du même type de capteurs.

3.2.3.1.1 Les données Une *donnée* est la spécification de l'évolution d'une valeur physique au cours du temps. Pour cela une *donnée* spécifie un flot de données en entrée de l'application. Le flot de données représenté par une *donnée* est caractérisé de la même manière que les flots de données des pilotes d'acquisition à laquelle on ajoute :

– son rôle, c'est-à-dire *Mesure* ou *Consigne* (*data_role*).

Le type de composant représentant les données dans SAIA est noté `data`. Son métamodèle est présenté figure 3.6. Une donnée possède une interface d'entrée *i_SetData* dont la spécification temporelle du flot qu'elle consomme est fournie par une entité de type *QoS_data*. Chaque

occurrence est consommée via l'appel au service fourni :

– `SetData(T_datatype)` //mise à jour de la donnée dans la plateforme abstraite

L'appel à ce service permet de mettre à jour les informations sur l'occurrence, contenue dans l'entité *T_ datatype*. Enfin, les composants de types *donnée* requièrent le service :

– `SetSoftData(T_datatype)` //mise à jour de la donnée dans l'application

afin de permettre au flot de données d'être consommé par l'application de contrôle.

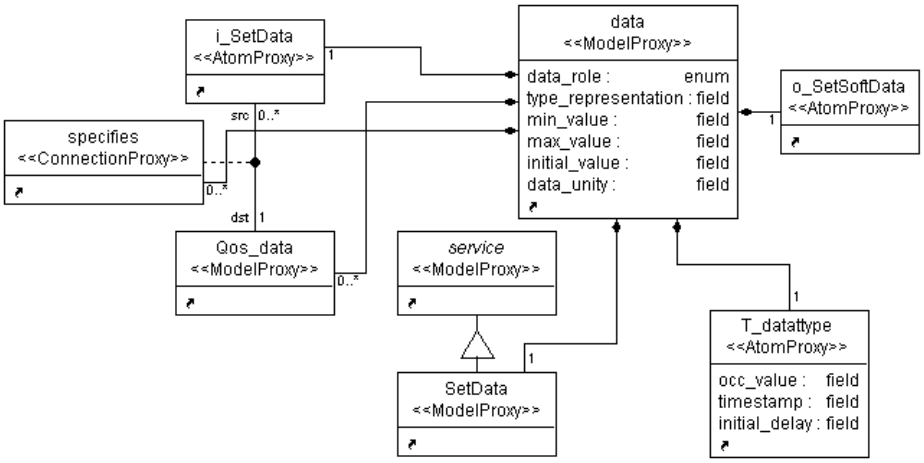


FIG. 3.6 – Métamodèle d'une donnée

3.2.3.1.2 Les événements Un *événement*, noté *Event* dans SAIA, est la spécification de l'évolution du changement d'un état physique. Pour cela un *événement* spécifie un flot d'événements en entrée de l'application.

Un *événement* fournit le service :

– `NewEvent(T_eventtype)` //arrivé d'un nouvel événement dans la plateforme abstraite

et requiert le service :

– `NewSoftEvent(T_eventtype)` //arrivé d'un nouvel événement dans l'application

Le changement de l'état physique spécifié par un événement est fonctionnellement caractérisé de la même manière qu'un pilote d'acquisition d'événements à laquelle on ajoute :

– son rôle, c'est-à-dire *Mesure* ou *Consigne* (*event_ role*).

3.2.3.2 Les sorties

Une *sortie* est une action abstraite sur le processus. Spécifier une *sortie* abstraite revient à spécifier directement la finalité de l'action que l'application doit entreprendre sur le processus. Il est important que cette action soit spécifiée de manière à refléter la finalité sur le processus plutôt que l'action en elle même pour éviter la spécification d'aspects liés à une technologie donnée. Par exemple, pour une application de contrôle d'un robot explorateur, il est tentant de spécifier une sortie du type "`freiner`". Cette action suggère la présence de frein, pourtant la finalité pour l'application est soit de ralentir le processus, soit de l'arrêter. On spécifie alors les sorties "`ralentir`" et/ou "`stopper`" plutôt que "`freiner`".

Comme pour les *entrées*, il est important de ne pas spécifier d'aspects technologiques lors de la spécification des *sorties* afin de pouvoir ajuster les *sorties* de la plateforme abstraite sur un maximum de pilotes de réalisation différents et donc un maximum de plateformes réelles différentes.

Il existe deux types de *sortie* différents, celles spécifiant un flot de données et celles spécifiant un flot d'événements. Elles sont respectivement nommées *commande* et *commande événementielle*. Chaque *sortie* est en charge d'un et un seul flot d'informations. Une *sortie* possède donc une seule interface d'entrée et une seule interface de sortie.

3.2.3.2.1 Les commandes Une *commande*, notée *cmd* dans SAIA, est la spécification d'une action sur le processus pouvant être paramétrée par une valeur. De ce fait une commande spécifie un flot de données en sortie de l'application.

Une *commande* fournit le service :
- `SetCmd(T_datatype)` //mise à jour d'une commande par l'application

et requiert le service :
- `NewSoftCmd(T_datatype)` //mise à jour d'une commande dans le connecteur complexe

Puisque les *commandes* représente un flot de données en sortie de l'application et à destination du processus, leur flot est caractérisé de la même manière que celui d'un pilote de réalisation de commandes.

3.2.3.2.2 Les commandes événementielles Une *commande événementielle* est la spécification d'une action non paramétrable sur le processus. De ce fait une *commande événementielle* spécifie un flot d'événements en sortie de l'application.

Une *commande événementielle* fournit le service :
- `NewEventCmd(T_eventtype)` //production d'une nouvelle commande événementielle par l'application .

et requiert le service :
- `NewSoftEventCmd(T_eventtype)` //arrivée d'une commande événementielle dans le connecteur complexe

Le type de composants représentant les commandes événementielles dans SAIA est noté `event_cmd`.

3.2.3.3 Le SAIM et la QoS

Afin de développer une application de contrôle de processus en se basant sur la description d'une plateforme abstraite, il est nécessaire de caractériser la QoS des flots d'informations sur les *entrées* et les *sorties* garantissant la stabilité du système et/ou la qualité de contrôle recherchée. Comme pour le SAM, chaque caractéristique de QoS peut être requise ou fournie.

Les *entrées* spécifient un flot d'informations à destination du système. Ce flot est donc consommé par l'application. Chaque *entrée* doit fournir une caractérisation du flot d'informa-

tions représentant la vue souhaitée du monde extérieur en termes de loi d'arrivée et de loi de retard sur l'acquisition des informations. Cette spécification est une exigence de l'application et définit donc la QoS requise sur les *entrées* pour garantir la stabilité du système.

Les *sorties* produisent un flot d'informations. Ce flot est produit par l'application. La loi d'arrivée est donc fonction de l'application. De ce fait, une *sortie* doit spécifier la loi d'arrivée en tant que QoS fournie.

Toujours pour les *sorties*, la loi de retard est la loi représentant les temps écoulés entre le moment où l'occurrence d'une *Commande* est générée et le moment où l'action correspondante sur le processus est considérée comme réalisée. Lorsque l'occurrence d'une *Commande* est générée, son retard est donc égal à 0. Le retard est principalement fonction de l'actionneur réalisant l'action ainsi que du processus recevant l'action. Ce temps n'est pas fonction de la manière dont est généré le flot d'informations des *sorties*. C'est une exigence de la plateforme abstraite à respecter par la plateforme réelle. La loi de retard sur les *sorties* est donc spécifiée en tant que QoS requise.

On voit donc que pour chacune des caractéristiques de QoS fournie dans le SAIM, on trouve la caractéristique de QoS requise correspondante dans le SAM et inversement. Ainsi, il est possible de gérer la satisfaction entre la QoS requise et la QoS fournie. La satisfaction fait l'objet de l'établissement d'un contrat de QoS réalisé par le connecteur complexe.

Afin de permettre de connecter la plateforme abstraite à un maximum de plateforme réelle, la QoS doit garantir la correction du système mais en décrivant la QoS de la manière la plus permissive possible.

3.2.3.4 Conclusion sur le SAIM

Le SAIM comprend des composants de types *application*, *entrées* et *sorties*. Les composants de types *entrées* et *sorties* servent d'interfaces de communication entre l'application et le monde extérieur. Ils fournissent et requièrent un ensemble de services qui spécifient une plateforme de manière abstraite. La mise en place de l'application de contrôle se base alors sur cette plateforme abstraite plutôt que sur une plateforme réelle. Ainsi, l'application de contrôle n'est pas dépendante des moyens technologiques grâce auxquels la communication avec le monde extérieur est effectivement réalisée.

Afin d'obtenir une spécification complète de la plateforme abstraite, une spécification de la QoS requise et fournie par cette plateforme abstraite est nécessaire. Sans cette spécification extra-fonctionnelle, la connexion de la plateforme abstraite et de la plateforme réelle ne peut pas garantir la correction de l'application de contrôle en termes de stabilité et/ou de qualité de contrôle.

Les *entrées* et les *sorties* ne peuvent pas être directement liées aux pilotes du SAM. D'une part les services ne sont pas identiques et d'autre part il est rare de pouvoir acquérir directement un élément physique particulier à l'aide d'un pilote d'acquisition ou de pouvoir réaliser la finalité d'une action physique particulière de manière directe par un pilote de réalisation ; plusieurs

opérations d'adaptation sont alors nécessaires. Ces adaptations sont réalisées par la "glue" du connecteur complexe décrit dans la section suivante.

3.2.4 Le connecteur complexe

Le connecteur complexe est formé d'un ensemble de sous connecteurs. Chacun d'eux est un composant composite, en charge des adaptations et mettant en relation soit des *entrées* avec des pilotes d'acquisition ; soit des *sorties* avec des pilotes de réalisation. De plus, les sous connecteurs peuvent communiquer entre eux si nécessaire afin de bénéficier des adaptations réalisées par les autres.

Afin de rester homogène avec la description de la plateforme abstraite et réelle, les composants d'adaptation sont différenciés selon qu'ils produisent un flot de données ou un flot d'événements. Afin de garder la différence entre données et événements, un composant d'adaptation est dédié à la production d'un type de flot particulier en sortie. Il peut cependant consommer indifféremment un flot de données ou d'événements en entrée. Le rôle d'un sous connecteur est :

- soit de prendre un ou plusieurs flots d'informations en provenance des pilotes d'acquisition et/ou d'autres sous connecteurs pour construire un ou plusieurs flots d'informations à destination des *entrées* ;
- soit de prendre un ou plusieurs flots d'information en provenance des *sorties* et/ou d'autres sous connecteurs pour construire un ou plusieurs flots d'informations à destination des pilotes de réalisation.

Pour ce faire, chaque connecteur possède une "glue" spécifiée par une configuration de composants. L'ensemble des connecteurs et de leur "glue" est spécifié par l'ALM (*Adaptation Layer Model*).

Les prochaine partie présente les différents types de sous connecteurs correspondant aux divers types d'*entrées* et de *sorties*. Ensuite, la partie suivante décrit les contraintes sur la structuration des sous connecteurs. Enfin, les types de composants spécifiant la "glue" de chacun des sous connecteurs sont présentés.

3.2.4.1 Les types de composants de l'ALM

SAIA distingue quatre types principaux de composants pour définir la plateforme abstraite dans le SAIM : les *données*, les *événements*, les *commandes* et les *commandes événementielles*. Chacun de ces types de composants possède un type particulier de composant dans l'ALM. Ces composants sont appelés composants d'adaptation (AE : *Adaptation Element*). On différencie donc quatre types de composants d'adaptations :

Les composants d'adaptation d'*entrées*

- les composants d'adaptation de données (DAE : *Data Adaptation Element*),
- les composants d'adaptation d'événements (EAE : *Event Adaptation Element*),

Les composants d'adaptation de *sorties*

- les composants d'adaptation de commandes (CAE : *Command Adaptation Element*),

– les composants d'adaptation de commandes événementielles (ECAE : *Event Command Adaptation Element*),

3.2.4.1.1 Les composants d'adaptation d'*entrées* Ils possèdent au minimum une interface d'entrée. Cette interface peut consommer un flot d'événements ou de données en provenance d'un pilote d'acquisition ou d'un autre composant d'adaptation. Ils consomment les flots d'informations en provenance des pilotes d'acquisition en fournissant les services suivants :

– `NewDriverData(T_datatype)` //mise à jour d'une donnée en provenance d'un pilote
– `NewDriverEvent(T_eventtype)` //arrivée d'un nouvel événement en provenance d'un pilote

Parallèlement, ils consomment les flots en provenance des autres composants d'adaptation en fournissant les services suivants :

– `NewAEData(T_datatype)` //mise à jour d'une donnée en provenance d'un sous connecteur d'*entrée*
– `NewAEEvent(T_eventtype)` //arrivée d'un nouvel événement en provenance d'un sous connecteur d'*entrée*
– `NewAECmd(T_datatype)` //mise à jour d'une donnée en provenance d'un sous connecteur de *sortie*
– `NewAECmdEvent(T_eventtype)` //arrivée d'un nouvel événement en provenance d'un sous connecteur de *sortie*

Chacun des composants d'adaptation d'*entrée* peut être en charge de l'acquisition d'une ou plusieurs *entrées*, il peut dont posséder une ou plusieurs interfaces de sorties. Les composants d'adaptation de données requièrent donc le service suivant :

– `SetData(T_datatype)` //mise à jour de la donnée dans la plateforme abstraite

et les composants d'adaptation d'événements requièrent le service suivant :

– `NewEvent(T_eventtype)` //arrivée d'un nouvel événement dans la plateforme abstraite

Toujours à propos des services requis, il faut noter que seuls les composants d'adaptation d'événements (EAE) peuvent produire des *événements*. Ainsi seuls les EAE requièrent le service `NewAEEvent`. De la même façon, seuls les composants d'adaptation de données (DAE) requièrent le service `NewAEData`. Ils fournissent cependant tous deux les services `NewDriverData`, `NewDriverEvent`, `NewAEData`, `NewAEEvent`, `NewAECmd` et `NewAECmdEvent` car un EAE peut construire un flot d'événements à partir d'un ou plusieurs flot(s) de données et inversement pour un DAE.

3.2.4.1.2 Les composants d'adaptation de *sorties* Ils possèdent les même caractéristiques que les composants d'adaptation d'entrées à la différence prêt qu'ils consomment les flots d'informations en provenance des composants de type *sortie* du SAIM. Pour cela ils peuvent fournir les services suivants :

– `NewSoftCmd(T_datatype)` //mise à jour d'une commande dans le connecteur complexe
– `NewSoftEventCmd(T_eventtype)` //arrivée d'une commande événementielle dans le connecteur complexe

Ils peuvent également consommer les flots d'informations en provenance des autres composants d'adaptation de *sortie* en fournissant les services :

– `NewAECmd(T_datatype)` //mise à jour d'une donnée en provenance d'un sous connecteur de *sortie*

– `NewAECmdEvent(T_eventtype)` //arrivée d'un nouvel événement en provenance d'un sous connecteur de *sortie*

Les composants d'adaptation de sorties peuvent posséder une ou plusieurs interfaces de sortie via lesquelles, après adaptation, ils fournissent des flots d'informations à destination des pilotes de réalisation. Les composants d'adaptation de commande requièrent donc le service :

– `NewDriverCmd(T_datatype)` // mise à jour d'une commande dans la plateforme réelle.

et les composants d'adaptation de commande événementielle requièrent le service :

– `NewDriverCmdEvent(T_eventtype)` //arrivée d'un nouvelle commande événementielle dans la plateforme réelle.

Comme pour les composants d'adaptation d'entrées, seuls les CAE peuvent produire des *commandes* pour les drivers de réalisation et donc seuls les CAE requièrent le service `NewAECmd`. De la même manière, seuls les ECAE requièrent le service `NewAECmdEvent`.

Remarque : Les composants du SAM et du SAIM possèdent un nombre d'interfaces imposé : une interface de sortie pour un pilote d'acquisition, une d'entrée et une de sortie pour les *Entrées* et les *Sorties* du SAIM, etc. Ces contraintes sont imposées par la spécification des arités entre le type de composant et ses interfaces. De plus, le type de chacune des interfaces est également imposé. Au contraire, les sous connecteurs peuvent posséder plusieurs interfaces d'entrée, de plusieurs types. Il est pourtant nécessaire de spécifier que chaque sous connecteur possède au minimum une interface d'entrée. Cette contrainte ne peut pas être imposée par la spécification des arités puisque chaque interface, prise à part peut intervenir zéro fois ou plus. Pour ce type de contrainte, l'ajout d'une contrainte OCL est alors nécessaire. La figure 3.7 illustre cette contrainte, placée dans le métamodèle sur les composants d'adaptation d'entrée.

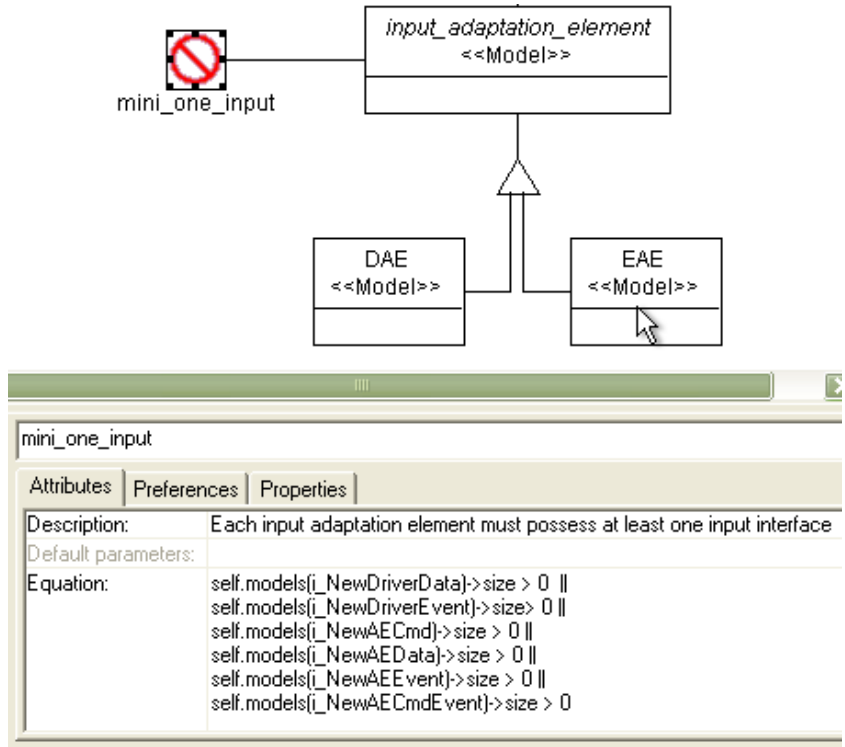


FIG. 3.7 – Une contrainte OCL pour assurer que les composants d'adaptation d'entrée possèdent au minimum une interface d'entrée.

3.2.4.1.3 Les interfaces de configuration Les composants d'adaptation peuvent nécessiter la description d'une ou de plusieurs constante(s) de configuration. On peut citer, par exemple, la réalisation de patron de conception "*observer*". Un "*observer*" surveille un flot de données et produit un *événement* sur une condition particulière des occurrences du flot comme, par exemple, lorsqu'une donnée dépasse un seuil prédéfini. Ce seuil, s'il n'est pas variable, est explicitement spécifié dans SAIA par une information de type constante. Ceci permet de rendre explicite certaines constantes de configuration qui introduisent un point de variabilité dans le comportement d'un sous connecteur. Afin d'être consultée par les sous connecteurs, ceux-ci peuvent posséder une ou plusieurs interfaces d'entrée de type :

– `AE_config`

3.2.4.1.4 Contraintes structurelles Les contraintes de structuration permettent d'imposer certaines règles sans lesquelles la description de la configuration est erronée. Dans SAIA, ces contraintes ont trois objectifs principaux :

1. réaliser les contrats de niveau 1 ;
2. imposer la fusion explicite des flots d'informations ;
3. interdire de "court-circuiter" le SAIM.

Chacun de ces objectifs et les contraintes associées sont expliquées par la suite.

1) Le premier type de contraintes de structuration permet de réaliser les contrats de niveau 1. Pour cela, elles doivent interdire la liaison entre des interfaces dont les services sont différents. Étant donné le nombre limité de services identifiés dans SAIA, on crée un type d'interface d'entrée et un type d'interface de sortie par service. Ainsi, dans le métamodèle, on ne permet la liaison qu'entre les interfaces d'entrée et de sortie se référant au même service. On obtient ainsi le métamodèle de la figure 3.8 auquel nous avons superposé un schéma des composants afin de montrer la structure adoptée.

2) De plus, sur le métamodèle de la figure 3.8, on exprime les arités acceptables entre chacune des interfaces fournies et chacune de celles requises. Ainsi, une interface d'entrée ne peut être connectée qu'à une et une seule interface de sortie. La connexion d'une interface d'entrée à plusieurs interfaces de sorties reviendrait à fusionner implicitement plusieurs flots d'informations. Dans SAIA, la fusion de différents flots d'informations doit être explicite et réalisée par la "glue" des composants d'adaptation car réaliser des fusions implicites ne permet pas d'établir les règles de fusion permettant de maîtriser la QoS caractérisant le flot résultant de la fusion (cf. section 3.3.3). De plus, pour les pilotes de réalisation, n'accepter qu'un seul flot en entrée permet de spécifier, au sein des composants d'adaptation de sortie (CAE et ECAE), les accès concurrents aux pilotes de réalisation et donc aux actionneurs.

Les contraintes imposant la fusion explicite de flots sont illustrées sur la figure 3.9 en prenant le cas des flots d'informations en entrée du système.

Malgré la contrainte précédente, les connexions point à point ne sont pas les seules acceptables dans SAIA. Comme les montre les arités de la figure 3.8, une interface de sortie peut être

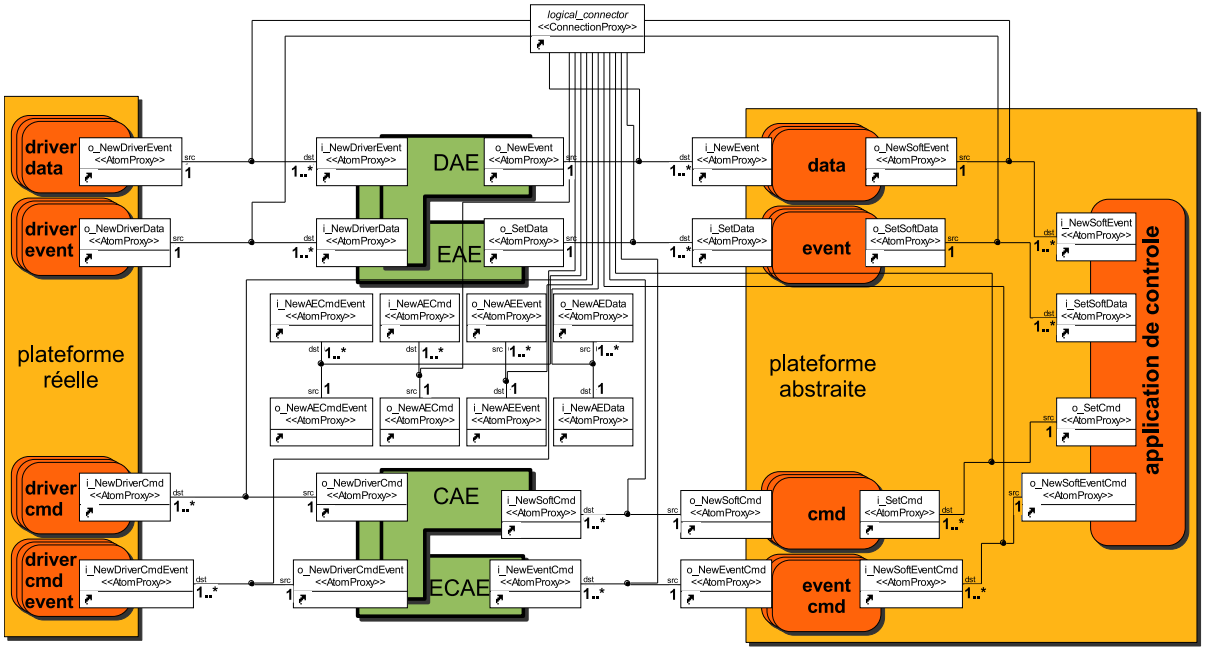


FIG. 3.8 – Métamodèle établissant la structuration et les arités entre les interfaces dans SAIA

reliée à une ou plusieurs interfaces d'entrée. Ce comportement correspond à un *broadcast* du flot d'informations depuis une interface de sortie vers toutes les interfaces d'entrées auxquelles elle est connectée. Comme nous avons vu que la communication d'un flot d'informations se fait par l'appel au service du composant consommateur du flot, le *broadcast* est donc effectué par une suite de `n` appels aux services des `n` composants associés. De ce fait, un ordre de précédence apparaît et il est donc nécessaire de préciser dans quel ordre les flots d'informations sont communiqués.

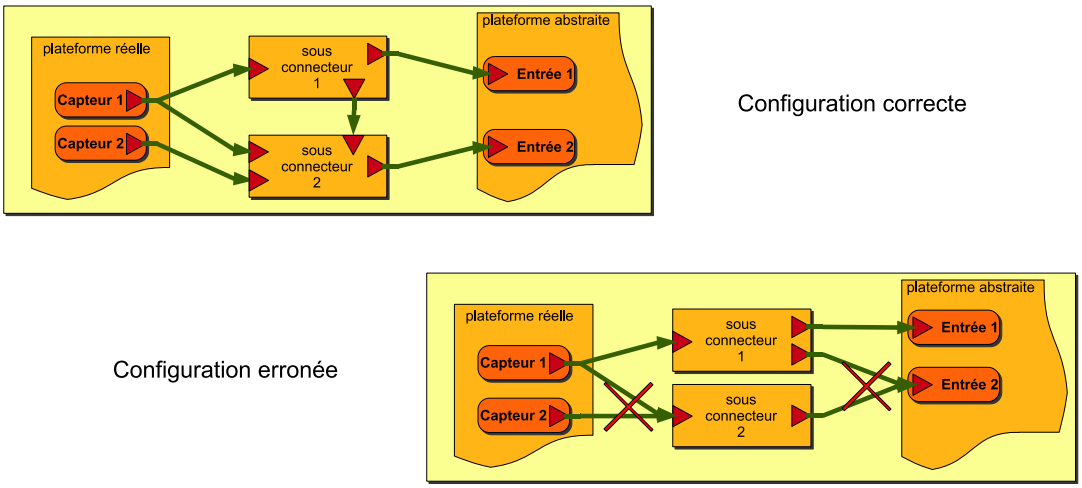


FIG. 3.9 – Exemple de structurations correctes et erronées dans SAIA

3) La dernière contrainte de structuration interdit aux composants d'adaptation de sortie de posséder les interfaces d'entrées et les services leurs permettant de recevoir les flots d'informations des composants d'adaptation d'entrée. Pourtant il est possible pour les composants

d'adaptation d'entrée de posséder les interfaces et les services leurs permettant de recevoir les flots d'informations des composants de sortie. Il est en effet possible d'effectuer une boucle ouverte des composants d'adaptation de sortie vers les composants d'adaptation d'entrées afin d'estimer une *entrée* d'après les *Commandes* envoyées au processus. Cependant, l'inverse revient à court-circuiter l'application de contrôle. En effet, envoyer les flots d'informations des composants d'adaptation d'entrée vers les composants d'adaptation de sortie implique que ces derniers possèdent une partie applicative spécifiant quelles sont les *Commandes* à envoyer au processus en fonction des informations en provenance du monde extérieur. Cette partie applicative doit impérativement être réalisée dans l'application de contrôle afin de ne pas être omise lors de la réutilisation du SAIM.

Chacun des types de composant d'adaptation est un composant composite possédant une "glue" afin de réaliser l'adaptation. Cette "glue" est réalisée par une configuration comprenant trois types de composants primaires détaillés dans la partie suivante.

3.2.4.2 Spécification de la "glue"

La "glue" des connecteurs d'adaptation permet de connecter les interfaces des pilotes du SAM avec les interfaces des *entrées* et des *sorties* du SAIM malgré leurs aspects hétérogènes. Dans SAIA, on distingue trois principales sources d'hétérogénéité, adressées chacune par un type de composant particulier. Ces types de composants particuliers sont ensuite connectés entre eux afin d'obtenir la "glue" désirée.

Les paragraphes suivants détaillent les sources d'hétérogénéité ainsi que les types de composants associés :
- Format : en charge de l'hétérogénéité de types / d'unités
- Interpret : en charge de l'hétérogénéité sémantique
- QoS_adapt : en charge de l'hétérogénéité de qualité de service.

3.2.4.2.1 Adaptation de types / unités

La première source d'hétérogénéité concerne les unités et/ou les types de représentation informatique d'une occurrence dans un flot. Puisque les unités et les types ne sont pas définis dans un flot d'événements, cette hétérogénéité ne concerne que les flots de données. Ce type d'hétérogénéité peut intervenir dans deux cas :
- lorsque les unités et/ou les types des occurrences d'un flot de données en entrée d'un connecteur sont différents de la spécification des occurrences du flot de données en sortie du connecteur.
- lorsque l'on désire fusionner deux ou plusieurs flots de données dont les unités et/ou les types des occurrences des flots sont différents.

Les changements d'unité ou de type nécessaires à une manipulation cohérente des occurrences de données sont à la charge du type de composant `Format` et constituent l'étape de formatage. Ce composant est toujours consommateur d'un et un seul flot de données et ne produit qu'un et un seul flot de données. De plus pour une occurrence du flot de données en entrée, il existe

toujours une et une seule occurrence du flot de donnée en sortie. Autrement dit, il n'y a ni filtrage ni duplication des occurrences dans ce composant. Le formatage est modélisé du point de vue temporel par un composant comprenant une fonction caractérisée par son pire et son meilleur temps d'exécution notés respectivement WCET et BCET (*Worst/Best Case Execution Time*).

Parmi les cas types, nous trouvons bien sûr le changement d'unité. Ce changement peut s'effectuer entre deux unités différentes d'une même grandeur, comme par exemple de degré Celsius vers degré Kelvin. Il peut également s'effectuer au sein de même unité comme par exemple lors du passage de mètre vers kilomètre. Les composants de type `Format` sont également utilisés lors d'un changement de repère. Encore une fois le changement peut s'effectuer de deux manières différentes : entre deux repères du même plan, ou entre des repères exprimés différemment (représentation cartésienne vers représentation polaire). Enfin, ces types de composants sont également utiles lors d'un changement de type, semblable à la notion de "cast" dans les langages de programmation.

3.2.4.2.2 Adaptation sémantique

Nous avons vu que les *entrées* du SAIM spécifient une vue abstraite d'une valeur physique nécessaire au contrôle et que les *sorties* spécifient une action abstraite sur le processus. Contrairement à cela, les pilotes d'acquisition fournissent une vue réelle du monde extérieur et les pilotes de réalisation réalisent une action réelle sur le processus. On peut donc avoir une hétérogénéité sémantique entre la spécification réalisée par les *entrées* et les *sorties* et la spécification des pilotes. Cette adaptation peut amener à fusionner ou scinder des flots d'informations.

Cette adaptation constitue la phase d'interprétation. Elle est réalisée par le type de composant `Interpret`. Puisque l'on peut fusionner ou scinder des flots d'informations, le composant `Interpret` peut posséder 1 à `n` flots d'informations en entrée et 1 à `n` flots d'informations en sortie. De plus, à une occurrence d'un flot d'entrée peut correspondre zéro, une ou plusieurs occurrences en sortie. Autrement dit, les composants de type `Interpret` peuvent filtrer ou dupliquer certaines occurrences en leur sein. Pour décrire le comportement des composants de type `Interpret`, on utilise un automate temporisé communicant et une fonction. L'automate temporisé communicant permet de spécifier la partie réactive de l'interprétation ; c'est-à-dire le lien entre la façon de consommer le(s) flot(s) en entrée du composant et la façon de produire le(s) flot(s) en sortie du composant. Parallèlement, la fonction spécifie les traitements à effectuer sur la ou les occurrences d'entrées afin de produire la ou les occurrences en sortie.

Nous donnons maintenant deux exemples permettant d'illustrer deux cas type de comportement d'un composant de type `Interpret`. Considérons dans le SAIM un composant de type *entrée*, modélisant `Vitesse_robot` qui spécifie la vitesse d'un robot. Considérons dans le SAM le pilote d'acquisition d'événements `Tick_roues`, générant un événement à chaque tour de roue. Il est possible de calculer l'*entrée* `Vitesse_robot` du robot en calculant la différence de temps entre deux occurrences successives du flot d'événements en provenance de `Tick_roues`. Il faut cependant réaliser une interprétation sémantique qui spécifie comment les événements de `Tick_roues` sont interprétés pour fournir l'entrée `Vitesse`. Dans ce cas l'automate peut, par exemple, spécifier si le calcul de la vitesse est réalisé à chaque nouvelle mesure de temps entre deux événements

ou est réalisé en faisant la moyenne de 2,3 ou `n` mesure(s) de temps entre deux événements. Parallèlement à cela, la fonction spécifie comment la moyenne est calculée à partir d'une ou de plusieurs mesure(s) de temps.

Un autre cas type, concerne la fusion de flot d'informations. Considérons maintenant, dans le contexte de l'exemple précédent, que le SAM possède deux pilotes d'acquisition d'événements `Tick_roue_droite` et `Tick_roue_gauche`, générant chacun un événement à chaque tour de la roue dont il est en charge. L'automate spécifie alors la manière de consommer et de produire les flots d'informations pour réaliser la fusion. Par exemple, il spécifie si l'*entrée* `Vitesse_robot` est calculée pour chaque occurrence de chacun des pilotes d'acquisition ou uniquement lorsque l'on a une nouvelle occurrence du pilote de `Tick_roue_gauche`. Ce ne sont bien sûr que des exemples et on peut réaliser de nombreuses autres spécifications de la partie réactive de l'interprétation lors d'une fusion de flots d'informations.

3.2.4.2.3 Adaptation de QoS

Nous avons vu que le SAM et le SAIM spécifient des caractéristiques de QoS requises et fournies. Des hétérogénéités peuvent apparaître au niveau de la QoS entre le SAM et le SAIM. Ces hétérogénéités peuvent apparaître au niveau de la loi d'arrivée et/ou de celle de retard.

L'adaptation permettant de pallier cette hétérogénéité est réalisée par les composants de type `QoS_adapt`. Le composant `QoS_adapt` est toujours consommateur d'un et un seul flot d'informations et ne produit qu'un et un seul flot d'informations du même type que celui d'entrée. Cependant, contrairement au composant `Format`, pour une occurrence du flot d'informations en entrée, il peut exister zéro, une ou plusieurs occurrence(s) en sortie. Les composants de type `QoS_adapt` peuvent donc filtrer ou dupliquer des occurrences de leur flot d'entrée. Comme pour le composant `Interpret`, `QoS_adapt` est défini par un automate temporisé communicant et une fonction.

Un des cas types est le filtrage. Par exemple, une *entrée* peut spécifier qu'elle requiert un flot d'informations ayant une loi d'arrivée périodique de 20ms. De l'autre côté, après formatage et interprétation, le flot d'informations originellement en provenance d'un pilote peut posséder une loi d'arrivée de 5ms. Dans cet exemple l'automate doit réaliser le filtrage des informations d'entrées en ne réémettant qu'une occurrence du flot d'entrée sur quatre. Parallèlement, la fonction peut spécifier que l'on émet la valeur maximum / minimum / moyenne des occurrences reçues entre deux réémissions.

Un autre cas type consiste à émettre périodiquement un flot d'informations non périodique. Si la période de réémission est inférieure à la loi d'arrivée du flot d'informations en entrée, la fonction du composant de type `QoS_adapt` peut alors réaliser une interpolation. Enfin, un autre cas type intéressant consiste à créer un retard constant sur les occurrences en sortie. Ce retard est, bien sûr, au minimum égal au retard maximum des occurrences du flot d'entrée, cependant, dans certains cas il est préférable d'avoir un retard plus important et constant que moindre et variable.

3.2.4.3 Conclusion sur l'ALM

L'ALM est le modèle d'une couche d'adaptation entre le SAM et le SAIM. Il définit un connecteur complexe composé de sous connecteurs pouvant communiquer entre eux. Chacune des *entrées* et des *sorties* du SAIM possède un type de sous connecteur spécifique défini par un composant composite. La configuration interne des sous connecteurs spécifie la "glue" nécessaire pour pallier à l'hétérogénéité entre d'un côté les interfaces des *entrées* et des *sorties* et de l'autre les interfaces des pilotes.

La "glue" est une configuration de trois types de composants : `Format`, `Interpret` et `QoS_adapt`. `Format` est seulement défini par une fonction alors que `Interpret` et `QoS_adapt` sont définis par une fonction et par un automate temporisé communicant. Il n'existe pas de contraintes particulières sur la structure de cette configuration.

Le connecteur complexe est une adaptation de la plateforme réelle définie par les pilotes du SAM afin qu'elle corresponde directement à la plateforme abstraite définie par les *entrées* / *sorties* du SAIM. Puisque le respect de la QoS spécifiée dans le SAIM est nécessaire au fonctionnement correct du système, le connecteur complexe est en charge de l'établissement d'un contrat de QoS entre le SAM et le SAIM. La section suivante détaille le mécanisme impliqué dans la mise en œuvre de ce contrat, en commençant par une définition formelle de la QoS considérée dans ce rapport.

3.3 SAIA et la QoS

3.3.1 Introduction

Nous avons vu que la QoS concernant la communication des applications de contrôle de processus avec le monde extérieur influence la stabilité et la qualité de contrôle du système. La littérature met en avant la loi d'arrivée et la loi de retard des informations. De plus, puisque la fusion entre des flots d'informations est possible dans le connecteur complexe, nous considérons également, dans le cadre de cette thèse, la loi de corrélation temporelle.

Nous avons également vu qu'une description de la QoS fournie et requise par les composants du SAM et du SAIM est réalisée dans SAIA afin de permettre l'établissement d'un contrat de QoS par le connecteur complexe.

Afin de pouvoir réaliser la description et la gestion de la QoS sous forme de contrat, plusieurs étapes sont nécessaires. Ces étapes sont décrites dans ce chapitre selon le plan suivant : La première partie décrit formellement les caractéristiques de QoS que sont la loi d'arrivée, la loi de retard et la loi de corrélation temporelle. La seconde partie explique l'impact du connecteur complexe et l'analyse qu'il nécessite. Enfin, la troisième partie explique la mise en place du contrat de QoS entre la plateforme abstraite et la plateforme réelle (cf. figure 3.2 chapitre 3.1).

3.3.2 Définition de la QoS

Les flots d'informations circulant dans le système sont en provenance ou à destination du monde extérieur. Nous faisons référence au monde extérieur comme à un ensemble d'informations et d'actions physiques. Un flot d'informations est alors la conséquence ou la cause d'une information ou d'une action physique. Un flot d'information est une suite infinie d'occurrence numérotée entre un et l'infini. Pour chaque occurrence d'un flot d'information on considère l'existence d'une information ou d'une action correspondante dans le monde extérieur. Une caractéristique de QoS est alors un modèle orienté QoS d'un flot d'informations. De ce fait chaque caractéristique de QoS est caractérisée par une suite infinie d'occurrences de QoS numérotées entre un et l'infini. Chacune des occurrences de QoS est liée à une occurrence dans le flot d'informations qu'elle caractérise. Dans la partie suivante, nous commençons par définir chacune des occurrences de QoS de manière informelle avant de présenter leur définition formelle.

3.3.2.1 Définition des occurrences de QoS

Trois caractéristiques de QoS sont considérées dans cette thèse : la loi d'arrivée, la loi de corrélation temporelle et la loi de retard. Avant de donner la définition formelle de chacune de ces occurrences, nous commençons par en donner une description informelle.

Informellement, une occurrence de la loi d'arrivée représente le temps écoulé entre l'arrivée de deux informations consécutives d'un même flot.

Une occurrence de la loi de corrélation temporelle représente la différence d'âge entre deux occurrences au moment de leur fusion. Cette loi n'existe donc que lorsqu'une fusion est réalisée dans le connecteur complexe.

Pour la loi de retard, on différencie la définition de la loi de retard d'une information entrante, notée $IDelay$, de celle d'une information sortante, notée $ODelay$. Une occurrence de la loi de retard $IDelay$ représente le temps écoulé entre le moment où l'information est présente dans l'environnement et le moment où l'occurrence est considérée dans le système. En sortie, une occurrence de la loi de retard $ODelay$ est informellement définie par le temps écoulé entre le moment où l'occurrence est générée dans le système et le moment où cette occurrence est considérée dans le système ou dans le monde extérieur. En automatique, lorsque le moment où l'occurrence est considérée correspond au moment où l'action correspondante sur le processus est réalisée, on appelle cette occurrence un temps de réponse. Comme classiquement en automatique, nous considérons une *Commande* comme réalisée lorsque l'action physique correspondante est réalisée à plus de 95%. Cette information est uniquement disponible dans le cas d'un système linéaire, ou d'un système non linéaire, linéarisé autour d'un point fixe [107].

Dans les définitions suivantes, nous donnons maintenant l'expression formelle de chacune des occurrences de QoS. Par la suite, un flot d'information f est une suite infinie d'occurrence f_i où $i \in [1;\infty]$. À un flot f dans le système correspond un flot φ dans le monde extérieur. Nous considérons donc que $\forall\ (f_i\ \in\ f)\ \exists\ (\varphi_i\ \in\ \varphi)$ tel que f_i représente φ_i. Une caractéristique de QoS, notée $QoSchar$, est une suite infinie d'occurrences $QoSocc_i$ où $i \in [1;\infty]$; $\forall\ (QoSocc_i\ \in$

$QoSchar$) $\exists$ ($f_i \in f$) tel que $QoSocc_i = QoS(f_i)$.

De plus, dans les définitions suivantes, $@f_i$ représente la date de création d'une occurrence f_i. De même, $@\varphi_i$ représente la date de création d'une occurrence φ_i. On note $f_i.init\_delay$, le retard initial de l'occurrence f_i. La définition formelle de $f_i.init\_delay$ est donnée par la suite. Puisque le retard d'une occurrence est dépendant du moment où on le considère, nous notons Tf_i la date à laquelle le retard de l'occurrence f_i est considéré. Lors d'une fusion de `N` flots d'informations, l'ensemble des flots fusionnés est noté F. Il comprend `N` flots notés fn où $n \in [1; N]$. (kp) représente le numéro de l'occurrence d'un flot fn ou d'un flot φn au moment de la k^{eme} fusion. L'occurrence correspondante est donc notée fn_{kp}. Enfin, nous notons :

$arrival_i$ est la $QoSocc_i$ associée à la loi d'arrivée de f_i ;

$IDelay_i^{Tf_i}$ est la $QoSocc_i$ associée à $IDelay$ de f_i au moment Tf_i ;

$ODelay_i^{Tf_i}$ est la $QoSocc_i$ associée à $ODelay$ de f_i au moment Tf_i ;

Δ_k est la $QoSocc_i$ associée à la loi de corrélation temporelle des occurrences fn_{kp} $\forall\ n \in [1; N]$.

Définition préalable : $f_i.init\_delay = @f_i - @\varphi_i \qquad \forall i \geq 1$

Les occurrences de QoS sont définies comme ceci (cf. figure 3.10) :

$$arrival_i = @f_i - @f_{i-1} \qquad \forall i \geq 1$$

$$IDelay_i^{Tf_i} = Tf_i - @\varphi_i \qquad \forall i \geq 1$$

$$ODelay_i^{Tf_i} = Tf_i - @f_i \qquad \forall i \geq 1$$

$$\Delta_k = \max_{1 \leq n,m \leq N} (|@fn_{kp} - fn_{kp}.init\_delay - @fm_{kp} + fm_{kp}.init\_delay|)$$
$$\forall n > 0;\ m > 0;\ k > 0\ fn, fm \in F$$

$$= \max_{1 \leq n,m \leq N} (|@\varphi n_{kp} - @\varphi m_{kp}|)$$
$$\forall n > 0;\ m > 0;\ k > 0\ @\varphi n_{kp} = QoS(fn_{kp}), @\varphi m_{kp} = QoS(fm_{kp})$$

Il est à noter que les occurrences de retard $IDelay_i^{Tf_i}$ sont exprimées par rapport au temps Tf_i et sont donc variable selon Tf_i. Il en est de même pour les occurrences de retard $ODelay_i^{Tf_i}$. Dans la suite de ce rapport, les temps Tf_i correspondent à des instants précis. Pour la caractéristique $IDelay$, Tf_i correspond au moment où l'occurrence est mise à jour dans l'entité où $IDelay$ est exprimée. Pour la caractéristique $ODelay$, Tf_i correspond au moment où l'action physique correspondante est considérée comme réalisée.

Enfin, la corrélation temporelle ne fait pas l'objet d'une spécification dans le SAIM ou le SAM. En effet, les valeurs acceptables pour la corrélation temporelle dépendent des types de

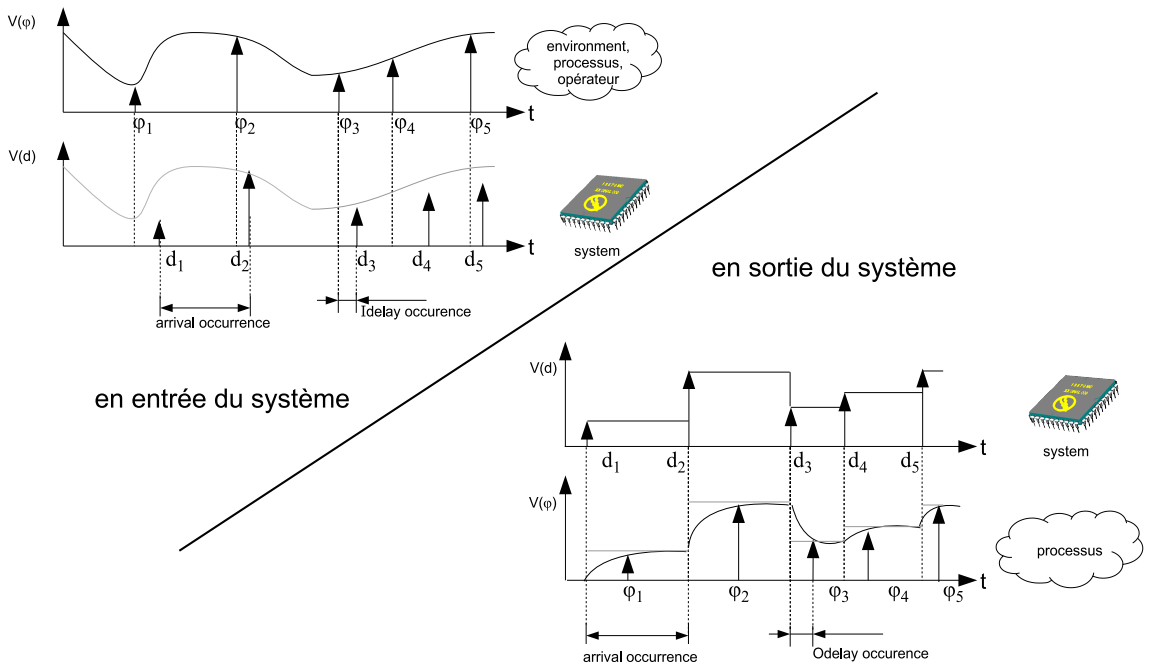


FIG. 3.10 – Représentation des occurrences de QoS

flots fusionnés et de leur dynamique. La présence d'une fusion ou non et des flots impliqués dans cette fusion n'est connue qu'au moment de la spécification du connecteur complexe. Cependant, cette caractéristique est importante pour assurer une certaine qualité de contrôle. La spécification des valeurs acceptables pour la corrélation temporelle s'effectue donc au moment où l'on réalise une fusion de flot d'informations dans le connecteur complexe. Cette caractéristique de QoS est donc spécifiée dans le composant primaire réalisant la fusion : `Interpret`.

Maintenant que les occurrences de QoS ont été définies, la section suivante définit les caractéristiques de QoS.

3.3.2.2 Définition des caractéristiques de QoS

Une caractéristique de QoS est la spécification d'une suite infinie de *QoSocc* d'un même type permettant de caractériser un flot f. Il existe plusieurs façons d'exprimer une suite infinie de *QoSocc*. La manière la plus simple est l'expression d'une constante, notée *cste*. Dans ce cas la spécification signifie que toutes les *QoSocc* doivent avoir la même valeur au cours du temps :

$$(QoS = cste) \iff (\forall i\ QoSocc_i = cste)$$

Pour une expressivité plus riche, une caractéristique de QoS peut être spécifiée par un intervalle. Dans ce cas, la spécification signifie que chaque *QoSocc* appartenant à la caractéristique peut appartenir au domaine de valeurs défini par l'intervalle.

$$(\min < \mathrm{MAX} \wedge QoS = [min\ ;\ MAX]) \iff (\forall i\ QoSocc_i \in [min\ ;\ MAX])$$

Dans ce cas, l'intervalle [1 ; 5] peut spécifier la suite {1, 5, 4, 3, 1, 4, 2, ...} mais aussi {1, 1, 1, 1, 1, ...} ou encore {1, 5, 1, 5, 1, 5, ...} etc.

Certaines applications de contrôle de processus peuvent nécessiter une spécification plus précise qu'un intervalle. Par exemple on peut souhaiter que chaque *QoSocc* appartienne à l'intervalle [1 ; 5] mais comme une suite de valeur unique ; c'est-à-dire {1, 1, 1, 1, ...} ou {2, 2, 2, 2, ...}. Dans ce cas, les intervalles ne sont pas assez expressifs. On propose alors d'utiliser des ω-expressions régulières afin de spécifier précisément une caractéristique de QoS. Une expression régulière permet de spécifier la grammaire d'un ensemble de mots. Dans ce cas les occurrences des caractéristiques de QoS ne peuvent prendre leur valeur que pour former un des mots possibles dans la grammaire spécifiée.

$$QoS = regular\ expression \iff \forall i > 1\ \{QoSocc_{i-1}, QoSocc_i\} \in regular\ expression$$

Puisque les flots d'informations caractérisés par la QoS sont infinis, ou plus exactement ont la durée de vie du système, l'expression régulière ne doit définir que des mots infinis. Nous utilisons donc les ω-expressions régulières. Ainsi, en plus des symboles de quantification classiques des expressions régulières, nous utilisons le symbole de quantification ω qui spécifie qu'il existe une infinité de répétitions de l'expression précédent le symbole. Ainsi, dans l'exemple précédent, la suite de valeur unique appartenant chacune à [1 ; 5] devient $[1^\omega \mid 2^\omega \mid 3^\omega \mid 4^\omega \mid 5^\omega]$. Contrairement à cela, la spécification d'un intervalle classique devient $[1 \mid 2 \mid 3 \mid 4 \mid 5]^\omega$ ou encore $[1 - 5]^\omega$.

Nous avons vu comment spécifier les caractéristiques de QoS. Il est à noter que la spécification est la même que la caractéristique soit requise ou fournie. Avant de voir comment établir le contrat de QoS, la partie suivante explique l'impact du connecteur complexe et l'analyse qui en résulte.

3.3.3 Analyse du connecteur complexe

3.3.3.1 Principes

La section 3.2.4 définit le connecteur complexe comme un ensemble de sous connecteurs spécifiés par des composants composites communicants. Chacun de ces composants composites comprend une "glue" qui est spécifié par la configuration émanant de l'interconnexion des composants primaires suivants : `Format`, `Interpret` et `QoS_adapt`. Ces composants primaires contiennent des fonctions représentées temporellement par leur WCET et leur BCET ainsi qu'un automate temporisé communicant pour `Interpret` et `QoS_adapt`. De plus, afin de prendre en compte les effets de l'implémentation, des budgets temporels représentant les temps de blocages dus à l'ordonnancement sont exprimés pour chaque sous connecteur.

La "glue" des sous connecteurs modifie le(s) flot(s) d'informations entre le SAM et le SAIM. Il est donc clair que la QoS des flots d'informations en entrée d'un sous connecteur est également modifiée par le comportement de ce sous connecteur. Autrement dit, chaque sous connecteur agit comme un modificateur de QoS. Un sous connecteur ne requiert aucune QoS particulière sur ces entrées. Cependant, une fois ses entrées connectées, il fournit une certaine QoS en sortie. Ainsi il est impossible de spécifier la QoS fournie par le sous connecteur sans connaître la QoS de ces flots d'entrée. De plus, étant donné le comportement complexe de la "glue" au sein de chaque sous connecteur (automate, fonction), la QoS n'est pas modifiée de façon triviale. Il est donc nécessaire d'évaluer la QoS résultante en sortie de chaque sous connecteur une fois la configuration interne spécifiant la "glue" réalisée et les interfaces d'entrée du sous connecteur connectées.

Ce constat nous amène à conclure qu'au sein d'un composant composite spécifiant un sous connecteur, la QoS considérée dans cette thèse est une propriété non directement composable (cf. chapitre 2.2.4). L'évaluation des caractéristiques de QoS passe donc par l'analyse de la configuration de chacun des sous connecteurs.

Pour effectuer l'analyse, on s'appuie sur la sémantique opérationnelle des sous connecteurs fournie par le langage d'automates temporisés IF. Cependant, certains choix restent encore à faire lorsque à une même caractéristique de QoS requise correspondent plusieurs caractéristiques de QoS fournie. Par exemple lors de la fusion de flots d'informations au sein d'un composant primaire `Interpret`. Ce choix concerne la sémantique de calcul de la QoS.

En effet, dans ce cas il existe plusieurs façons de calculer la QoS résultante en sortie du connecteur. Par exemple, considérant un automate qui réalise la fusion de deux flots de données et qui produit une nouvelle occurrence de donnée uniquement lorsqu'une nouvelle occurrence de chaque flot de données en entrée a été reçue. Dans SAIA, le retard de la donnée fusionnée peut, au choix, être le retard maximum des occurrences au moment de la fusion (cf. figure 3.11 A), le retard minimum des occurrences au moment de la fusion (cf. figure 3.11 B), une moyenne des retards des occurrences au moment de la fusion (cf. figure 3.11 C).

Ce choix est souvent fonction de la dynamique des flots d'informations fusionnés et est donc laissé à la charge de l'utilisateur. Cependant, il est nécessaire que ce choix soit renseigné explicitement afin de permettre l'évaluation de la QoS résultante en sortie du connecteur. Une fois ce choix renseigné, l'évaluation proprement dite peut commencer.

3.3.3.2 Méthode d'analyse

L'analyse permettant l'évaluation de la QoS est effectuée sous connecteur par sous connecteur. La première étape consiste à isoler les composants dont le flot d'informations entre dans le sous connecteur. Pour chacun de ces composants, il existe une description de la QoS. La deuxième étape consiste à traduire la QoS exprimée sous forme d'expressions régulières en automate IF. L'automate IF doit reproduire un flot d'informations conforme aux spécifications réalisées par chacune des caractéristiques de QoS. La deuxième étape commence donc par une conversion de l'expression régulière sous forme d'un LTS déterministe [13]. Un LTS est un graphe d'état et de transition et où chaque transition contient une étiquette. Une fois le LTS réalisé, il faut construire un automate IF contenant autant d'état que le LTS. Chacune des étiquettes sur les

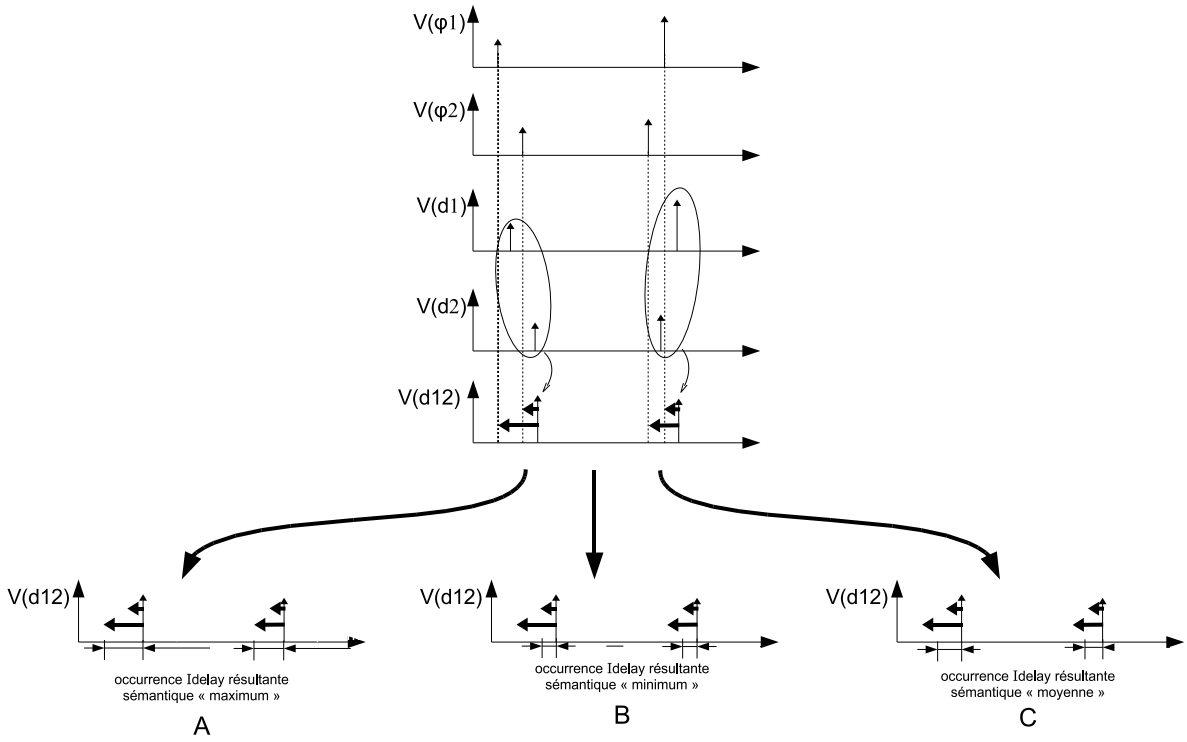


FIG. 3.11 – Exemple de sémantique de la loi de retard lors de la fusion de deux informations

transitions sortantes d'un état du LTS correspond alors à une garde temporelle dans l'automate IF. Cet automate est un simulateur du comportement temporel d'un des composants dont le flot d'informations entre dans le sous connecteur. Puisque nous nous intéressons uniquement à des propriétés temporelles, les valeurs associées à un flot (dans le cas d'un flot de données) ne sont pas utiles, seul le comportement temporel du flot est important.

Une fois les automates IF représentatif des expressions régulières réalisés, la partie représentant le sous connecteur et ces entrées constitue un système IF exécutable par les outils associés au langage. L'exécution d'un système IF réalise une simulation exhaustive du système aboutissant à la création d'un LTS.

Le LTS IF contient la combinatoire de tous les chemins d'exécution possibles du système (ici du sous connecteur). Afin de connaître la caractérisation du ou des flot(s) d'informations en sortie du système IF, l'exécution doit être observée. Dans notre cas, la mesure des occurrences de QoS $QoSocc_i$ correspond à une propriété de vivacité bornée. En effet, une occurrence de la loi d'arrivée met en relation deux occurrences consécutives dans un flot infini. Puisque le flot est supposé infini, pour chaque occurrence 'i', il existe une occurrence 'i+1'. La mesure de $arrival_i$ est donc une propriété de vivacité bornée. En ce qui concerne une occurrence $delay_i$, son calcul met en relation des informations φ_i du monde extérieur avec des informations f_i dans le système ou inversement. Précédemment, nous avons fait la supposition que pour chaque f_i dans le système il existe un φ_i dans le monde extérieur et donc un $QoSocc_i$. Il s'agit donc encore d'une propriété de vivacité bornée. Pour ce type de propriétés, il est donc possible de réaliser cette mesure grâce à des observateurs IF, un artefact du langage noté `observer`.

Un `observer`, non intrusif dans notre étude, peut déclencher des actions prioritaires sur l'envoi ou la réception d'événements dans le système. On propose de les utiliser pour mesurer les occurrences de QoS *QoSocc*. Lorsqu'une `QoSocc` est mesurée par l'`observer`, il génère un message. Ce message est alors traduit par une transition dans le LTS résultant de l'exécution du système. L'étiquette associée à la transition correspond alors à la valeur de l'occurrence de QoS mesurée.

Le LTS résultant de l'observation du système par les `observer` contient la combinatoire de toutes les exécutions possibles du système ainsi que toutes les valeurs possibles des occurrences de QoS correspondantes. Deux étapes sont alors nécessaires à l'extraction de l'ensemble des occurrences de QoS représentant la caractéristique de QoS mesurée. Premièrement, afin de ne pas cacher de comportement non désirable du système, le LTS résultant de l'exécution du système doit être cyclique et ne posséder aucun état bloquant (*deadlock free*).

Une fois ces propriétés vérifiées, il est possible de cacher les transitions non générée par les `observer` afin de ne garder que les *QoSocc*. Il en résulte un LTS contenant les *QoSocc* et des ϵ-transitions [4]. Il est alors possible de réduire le LTS obtenus selon la relation d'équivalence faible de trace [117] afin de ne garder que les *QoSocc*. Le LTS résultant de ces deux opérations ne contient alors que l'enchaînement des *QoSocc* représentatif de l'expression régulière de la caractéristique de QoS mesurée.

Ces étapes permettent de mesurer la QoS en sortie d'un sous connecteur malgré sa modification non triviale par la "glue". Ces étapes doivent être réalisées pour chaque sous connecteur. Ensuite, l'étape suivante consiste à vérifier que la plateforme réelle, adaptée, peut être liée à la plateforme abstraite tout en satisfaisant la QoS exprimée pour la correction du système. Autrement dit, la prochaine étape consiste à vérifier si le contrat de QoS entre la plateforme réelle et la plateforme abstraite peut être établit.

3.3.4 Établissement du contrat de QoS

Établir un contrat de QoS doit donc assurer que la connexion du SAM avec le SAIM réalise un système dont la qualité de contrôle est celle recherchée. Ceci revient à s'assurer que chacune des caractéristiques de QoS fournie satisfait la caractéristique de QoS correspondante lors de la connexion du SAM et du SAIM.

Plus précisément, un contrat est établi pour chaque caractéristique de QoS de chaque *entrées* et de chaque *sorties*. L'ensemble de ces contrats, s'ils sont tous établis correctement, permet l'établissement d'un contrat global. Pour la suite, on introduit l'opérateur `satisfies` qui vérifie que la satisfaction de la QoS est atteinte pour une caractéristique de QoS d'une *entrée* ou d'une *sortie* donnée. Cet opérateur est booléen : la QoS est satisfaite ou non. L'établissement d'un contrat de QoS consiste donc à vérifier :

$$QoS_{fournie}\ \text{“}satisfies\text{”}\ QoS_{requise}$$

La QoS est exprimée par deux caractéristiques, chacune d'elles étant exprimée comme un flot infini d'occurrences de QoS notées $QoSocc_i$. Le but de l'opérateur `satisfies` est donc de vérifier que pour tout i, chacune des $QoSocc_i$ spécifiée par la QoS fournie est spécifiée comme acceptable par la QoS requise.

La façon de réaliser l'opérateur `satisfies` est différente suivant la méthode choisie pour spécifier les caractéristiques de QoS. Si les caractéristiques sont spécifiées sous forme d'une constante, l'opérateur `satisfies` doit s'assurer que la constante représentant la QoS fournie est égale à celle représentant la QoS requise :

$$QoS_{fournie} = QoS_{requise}$$

Si les caractéristiques de QoS sont spécifiées par un intervalle, alors la relation qui permet de vérifier que la QoS fournie est composée exclusivement de `QoSocc` spécifiées comme acceptables par la QoS requise est la relation d'inclusion : $\subseteq$. Ainsi, chacune des valeurs possibles pour les *QoSocc* de la QoS fournie appartient aux valeurs acceptables spécifiées par la QoS requise, la relation est donc la suivante :

$$QoS_{fournie} \subseteq QoS_{requise}$$

Enfin, si les caractéristiques de QoS sont spécifiées par des expressions régulières, chaque caractéristique forme un langage. La QoS requise décrit un langage dont les mots infinis spécifiés décrivent les suites de valeurs acceptables. Parallèlement, la QoS fournie décrit les suites de mots infinis possibles, en sortie de la plateforme.

Si la QoS fournie est un sous langage de la QoS requise alors tous les mots de la QoS fournie font partie du langage de la QoS requise. L'opérateur `satisfies` doit donc vérifier la relation de sous langage entre la QoS fournie et la QoS requise. La relation de sous langage est vérifiée sur les LTS. En effet, l'évaluation de la QoS en sortie de chaque sous connecteur est déjà sous forme de LTS, de plus, La traduction d'une expression régulière vers un LTS se fait de manière assez naturelle. Il faut cependant s'assurer que le LTS représente la totalité des mots décrits par l'expression régulière. Pour les expressions régulières complexes, il est intéressant de réaliser un automate IF afin qu'il génère le LTS lui même. Une fois chacune la caractéristiques de QoS fournie et la caractéristique de QoS fournie correspondante toutes deux obtenues sous forme de LTS, la relation de sous langage peut être vérifiée.

La relation de sous langage doit vérifier que tous les mots possible dans la description du LTS d'une caractéristique de QoS fournie existent dans le LTS de la caractéristique de QoS requise correspondante. Plus précisément, la suite de transitions, et donc d'étiquettes du LTS d'une caractéristique de QoS fournie doit exister dans le LTS de la caractéristique de QoS requise correspondante. Cette relation est appelée "équivalence de trace" [117].

Ainsi l'opérateur "satisfies" doit, dans ce cas vérifier l'équivalence de trace entre la QoS fournie et la QoS requise. Nous notons cet opérateur *sublanguage_ of*. La relation devient :

$QoS_{fournie}$ "$sublanguage\_of$" $QoS_{requise}$

L'équivalence de trace est une relation implémentée dans l'outil aldébaran [47] et peut donc être vérifiée automatiquement. Il faut remarquer qu'une valeur constante `C` est un cas particulier d'un intervalle [`C1` ; `C2`] où `C1` est égal à `C2`. De même, un intervalle [`C1` ; `C2`] est un cas particulier d'un langage décrit par l'expression régulière [`C1` | `C1+1` | `C1+2` | ... | `C2`]$^{\omega}$

3.3.4.1 Résumé des étapes d'établissement d'un contrat de QoS dans SAIA

Les sections précédentes décrivent en détail la QoS en termes de descriptions formelles, d'analyses et de relations de satisfaction. Plusieurs étapes sont nécessaires pour permettre l'établissement d'un contrat de QoS dans SAIA. Ces étapes sont résumées dans ce chapitre. La description des caractéristiques de QoS requises et fournies n'est pas symétrique entre les *entrées* et les *sorties* (cf. section 3.2.3.3). Les étapes nécessaires à l'établissement d'un contrat de QoS sont donc légèrement différentes, nous les présentons dans la même section mais au travers de deux illustrations différentes.

Afin d'établir un contrat de QoS, il est nécessaire de posséder une description, sous forme de LTS des caractéristiques de QoS requises et des caractéristiques de QoS fournies, modifiées par le sous connecteur en charge de l'établissement du contrat de QoS. L'obtention des LTS représentants les caractéristiques de QoS requises et les caractéristiques de QoS fournies peut se faire de manière parallèle. Nous commençons arbitrairement par l'obtention des caractéristiques de QoS fournie, modifiées par le sous connecteur.

La première étape, notée 1 sur les figures 3.12 et 3.13, consiste à traduire la QoS fournie de chacun des composants liés au sous connecteur à analyser en un automate IF. Chaque automate doit produire un flot d'informations dont le comportement temporel est conforme à celui décrit dans les ω-expressions régulières de la QoS. Une fois cette étape réalisée, il convient d'extraire les automates IF de chacun des composants de la "glue" du sous connecteur considéré (étape notée 2 sur les figures 3.12 et 3.13). Il faut ensuite ajouter les *observers* de QoS ; si pour une même caractéristique de QoS requise correspondent plusieurs caractéristique de QoS fournie, il faut choisir l'*observer* conforme à la sémantique de calcul de la QoS. Cette étape est notée 3 sur la figure 3.12 et sur la figure 3.13. On réalise ensuite l'exécution de l'ensemble des automates IF (étape 4 de les figures 3.12 et 3.13) afin d'obtenir le LTS contenant la combinatoire de tous les chemins d'exécution possibles pour le comportement du sous connecteur, auquel est ajouté la mesure des occurrences de QoS effectuée par l'*observer*. La dernière étape pour l'obtention du LTS de la QoS fournie, modifiée par le sous connecteur est notée 5 sur la figure 3.12 et sur la figure 3.13. Elle consiste à effectuer les réductions décrites dans la section précédente afin de ne garder, dans le LTS de départ, que la description du langage de la QoS, sous forme de LTS.

Afin d'établir un contrat de QoS est également nécessaire de posséder le langage de la QoS requise sous forme de LTS. Pour ce faire trois étapes sont nécessaires. La première, notée `I` sur les figures 3.12 et 3.13, consiste à traduire la QoS requise par l'*entrée* en un automate IF. Cet automate doit produire l'ensemble mots de QoS conforme à l'ensemble des mots décrits dans les

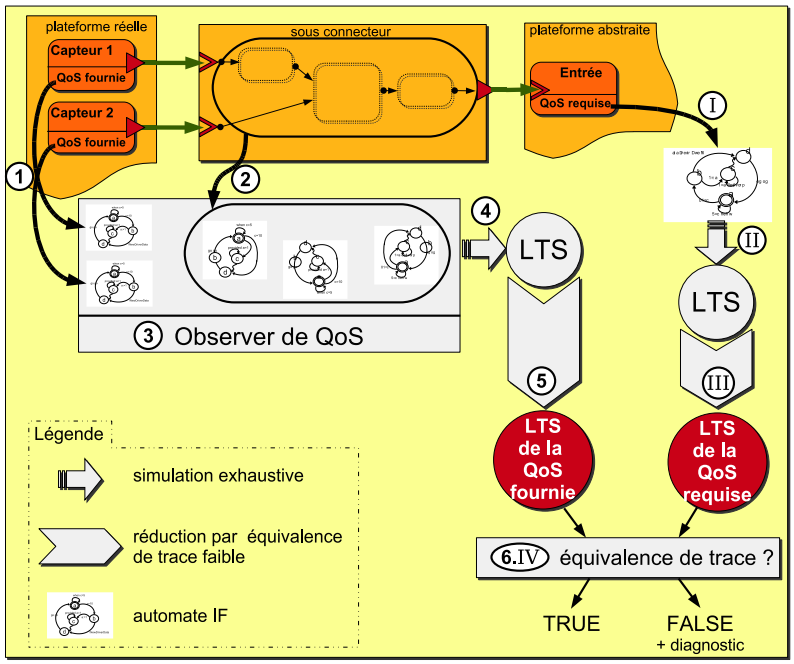


FIG. 3.12 – Étapes pour l'établissement d'un contrat de QoS d'une *entrée*

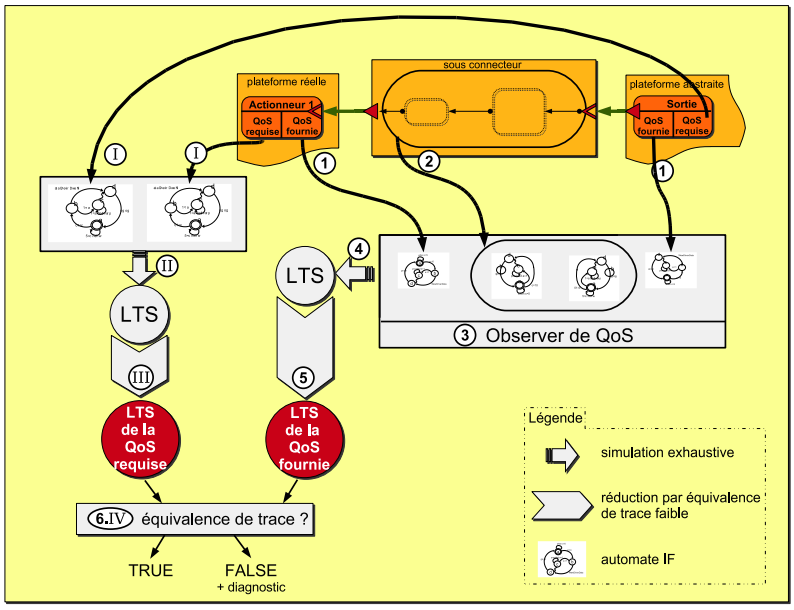


FIG. 3.13 – Étapes pour l'établissement d'un contrat de QoS d'une *sortie*

ω-expressions régulières de chaque caractéristique de QoS requise. L'étape suivante (notée `II` sur les figures 3.12 et 3.13) consiste à faire la simulation exhaustive du ou des automate(s) obtenu(s) à l'étape précédente. Enfin, l'étape notée `III` sur les figures 3.12 et 3.13 permet de réduire le LTS obtenu à l'étape précédente afin de ne conserver que le langage de la QoS requise, exprimé sous forme de LTS.

Une fois ces étapes réalisées, il est possible de vérifier la relation de sous langage en appliquant la relation d'équivalence de trace entre le LTS de la QoS fournie et celui de la QoS requise. Cette relation, notée 6.`IV` sur les figures 3.12 et 3.13, renvoie un résultat booléen. Si ce résultat est `vrai`, le contrat peut être établi. S'il est `faux`, l'outil aldébaran utilisé pour vérifier la relation d'équivalence fourni un diagnostic qui précise le mot de QoS jusqu'à la première occurrence de QoS fautive.

3.4 Conclusion

Ce chapitre a présenté les principes du style architectural SAIA, c'est à dire une séparation des préoccupations entre le côté applicatif d'un système de contrôle de processus et la partie plateforme de communication avec le monde extérieur.

Pour cela, l'application de contrôle est basée sur la description d'une plateforme abstraite décrite en termes d'*entrées* et de *sorties*. Le contrôle du procédé est assujetti à une qualité de contrôle, exprimant la précision et/ou la stabilité requise pour le système. Dans SAIA, la qualité de contrôle est dérivée en QoS sur la plateforme abstraite.

Afin de réaliser le système, une plateforme réelle doit être choisie et analysée afin d'en connaître la QoS. Elle est ensuite connectée à la plateforme abstraite par un connecteur complexe. Ce connecteur complexe est composé de sous connecteurs pouvant communiquer les uns avec les autres. Ils sont en charge de l'adaptation de la plateforme réelle afin qu'elle corresponde directement à la plateforme abstraite. Pour cela une "glue" est spécifiée pour chaque sous connecteur sous la forme d'une configuration de composants.

La structuration d'une architecture conforme à SAIA est soumise à des contraintes sur les associations entre les types de composants. Les contraintes sont principalement dues à la nécessité de réaliser des analyses. Par exemple, le fait qu'une interface ne puisse être liée qu'à une et une seul interface de sortie a été introduit pour éviter les fusions implicites de flots d'informations. Cette fusion implicite n'est gênante que par le fait qu'elle ne permet pas de connaître la sémantique d'agrégation des flots d'informations et donc ne permet pas la réalisation d'analyses de QoS. Les agrégations ont donc été isolées dans un composant particulier et définies sous la forme d'un automate temporisé communicant. En ce sens, les restrictions sont placées de manière à permettre à une configuration d'être analysable.

Toujours dans l'idée de réaliser des analyses sur la QoS des flots d'informations circulant entre l'environnement et le système, une séparation est faite entre les flots de données et les flots d'événements. Pourtant, au regard des analyses réalisées, aucune d'elles n'exploite cette

différence. Il nous semble tout de même important de faire la différence entre les flots d'événements et les flots de données afin de permettre des extensions futures de SAIA telles que l'ajout d'analyses supplémentaires. En particulier, sur la précision des données et l'impact des retards, des agrégations ou autres adaptations telles que les interpolations sur cette précision.

Les analyses proposées permettent d'évaluer l'impact du connecteur complexe et ainsi de vérifier la correction du système à l'aide d'établissements de contrats de QoS. L'établissement d'un contrat reflète la satisfaction d'une caractéristique de QoS requise par la caractéristique de QoS fournie correspondante. Lorsqu'elle concerne les services d'un composant, la notion de requis et de fourni est relativement intuitive. Cependant pour ce qui est de la QoS, elle est moins évidente. Par exemple, la loi d'arrivée d'un pilote d'acquisition est définie dans SAIA comme une caractéristique fournie parceque le pilote génère le flot d'informations ainsi caractérisé. Cependant, à l'image du débit entre un client et un serveur, on pourrait admettre que la loi d'arrivée d'un pilote d'acquisition est une loi requise afin que le consommateur du flot puisse consommer chacune des occurrences du flot. Dans SAIA, deux règles ont permis d'établir si les caractéristiques étaient fournies ou requise. D'abord, puisque le but premier est de caractériser la QoS permettant à l'application d'atteindre la qualité de contrôle désirée, le point de vue de l'application est un principe directeur. Ainsi la loi d'arrivée d'une *entrée* est une caractéristique requise pour la stabilité. La deuxième règle est venue d'un besoin de cohérence dans la relation de satisfaction permettant l'établissement d'un contrat. Ainsi, non seulement pour chaque caractéristique de QoS fournie il existe une caractéristique de QoS requise mais également la relation de satisfaction doit toujours exprimer la notion de sous langage (lorsque la caractéristique est exprimée par une expression régulière).

Enfin, avant de conclure ce chapitre, il me semble intéressant de se pencher sur les budgets temporels introduits dans SAIA. SAIA ne fournit pas d'éléments permettant la description de la plateforme d'exécution réelle. La plateforme d'exécution est abstraite par la spécification des différents budgets temporels au sein de chacun des composants. Un budget temporel est une abstraction temporelle des services d'exécution. Cette abstraction est nécessaire (et suffisante) à la réalisation des analyses présentées. Cependant, puisque SAIA ne fournit pas de moyen de décrire la plateforme d'exécution réelle, elle ne spécifie pas non plus comment construire le lien entre cette abstraction et sa réalité. De ce fait, SAIA reste une architecture logique.

Nous avons vu dans ce chapitre les principes de SAIA. Le chapitre suivant se propose d'évaluer la pertinence et le cadre d'utilisation de la proposition. À cet effet il présente les différentes utilisations et mises en œuvre de SAIA.

4

Évaluation de l'approche

Sommaire

Ce chapitre a pour but d'évaluer le cadre d'utilisation de SAIA. Pour ce faire, le chapitre commence par la description de l'outil de modélisation réalisé pour supporter SAIA. Ensuite, une méthode est proposée pour l'utilisation de SAIA. Cette méthode est illustrée par un exemple simple mettant en évidence les intérêts de l'approche, en particulier pour la gestion de la QoS. Une deuxième utilisation de l'approche est alors illustrée, également à l'aide d'un exemple simple. Ensuite, l'utilisation est présentée pour la modélisation et pour l'implémentation d'un robot explorateur. Enfin, autre exemple de robot explorateur présente la réutilisation du modèle SAIM du robot précédent dans un autre contexte. Chaque partie de ce chapitre propose une critique de l'approche.

4.1 Outil de modélisation

Respecter un style architectural demande une grande connaissance du style en question. Chacun des types ainsi que chacune des contraintes doivent préalablement être connus. De plus, il faut une grande attention afin de ne pas transgresser involontairement une des contraintes énoncées par le style. La réalisation d'un outil de modélisation spécifique au style architectural permet alors de s'assurer du respect des types de composants ainsi que des contraintes sur leur association. Pour SAIA, un outil basé sur GME [62] a été développé. GME est un outil pour la création d'environnement de développement spécifique. À cette fin, il propose de modéliser le langage cible sous forme d'un métamodèle basé sur une extension d'UML. Le but de l'outil réalisé est de fournir un éditeur de modèles conformes à SAIA. Il permet de modéliser chacun des types de composants et de modèles présentés dans cette thèse. Il implémente également chacune des contraintes structurelles présentées.

La description des métamodèles de SAIA a donc été nécessaire. Cette étape permet de poser clairement les concepts employés. Une fois les métamodèles de SAIA représentés dans l'outil, des informations de présentation doivent être ajoutées aux métamodèles afin de générer l'outil de modélisation de SAIA. Les informations de présentation permettent la description de plusieurs vues où les types de composants sont utilisables et/ou visualisables ou non. Pour SAIA, chacun des modèles (SAIM, SAM ou ALM) est réalisé dans l'outil par une vue différente. Ceci permet de ne proposer que l'utilisation des types de composants relatifs au modèle en cours de réalisation. Les informations de présentation permettent également de définir différentes vues selon la profondeur où l'on se trouve dans le modèle (La profondeur étant, ici, liée à l'arborescence dans les composants composites). Ainsi, lors de la modélisation du comportement d'un composant d'adaptation (xAE), seul les composants spécifiques à la "glue" sont utilisables. Ceci permet de séparer les préoccupations relatives à chacun des modèles mais aussi celles relatives à la profondeur où l'on se trouve dans le modèle. De plus, afin de séparer les aspects fonctionnels et extra-fonctionnels, la description de la QoS est réalisée dans une vue différente de la description du comportement de chacun des types de composant.

Une boîte à outils proposant des composants prédéfinis est proposée dans l'outil réalisé. Cette boîte à outils permet d'utiliser directement un comportement particulier, comme par exemple un type d'agrégation spécifique. Ceci permet de gagner du temps pour la réutilisation d'un comportement fréquemment rencontré. Cela permet aussi d'appréhender directement le code IF exécutable lié à ce comportement.

La figure 4.1 est une capture d'écran de l'outil permettant la réalisation de modèles SAIA. L'onglet sélectionné est celui du SAIM, Les types de composants utilisables ne sont donc que ceux appartenant au SAIM. Les attributs des composants sont accessibles dans la fenêtre en bas à droite. Pour les composants composites, il est possible de décrire leur configuration interne en "rentrant" dans le composant composite ; une nouvelle page d'édition de modèles s'ouvre alors. Enfin, tous les composants et les modèles sont accessibles via l'explorateur se trouvant à la droite de la zone de modélisation.

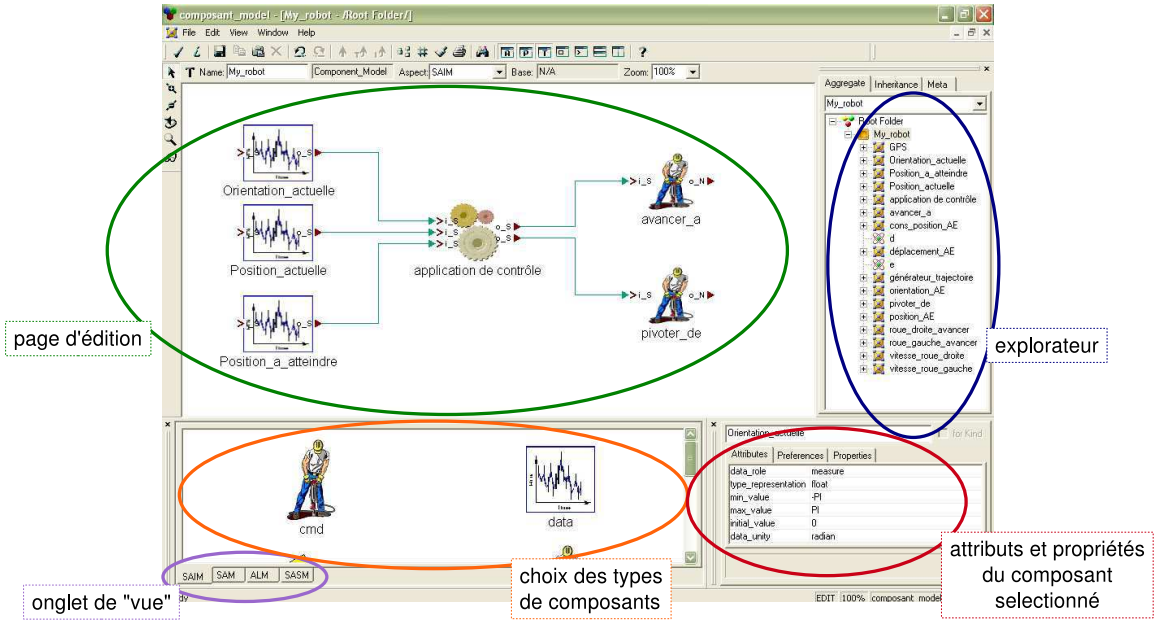


FIG. 4.1 – Capture d'écran de l'outil réalisé

Une fois la modélisation d'un système réalisé dans l'outil, il faut débuter le processus d'établissement des contrats de QoS. Nous avons vu section 3.3.3 qu'il est nécessaire pour cela de réaliser des analyses. Ces analyses sont implémentées par les outils de manipulation de LTS associé à IF [17]. Afin de permettre l'utilisation des outils IF existant, un *parser* a été réalisé pour extraire un modèle IF d'une représentation SAIA. Le *parser* exploite les facilités de GME qui permet de convertir automatiquement les modèles SAIA en XML. Une fois les informations du fichier XML récupérées, elles sont compilées de manière à obtenir un modèle IF exécutable.

L'outil de modélisation développé ne capture pas tous les concepts manipulés par le langage IF (le but n'étant pas ici de fournir un éditeur IF). Afin de générer un modèle IF exécutable, il est donc nécessaire d'utiliser les comportements prédéfinis associés à la boîte à outils. De plus, le *parser* permettant la traduction du modèle SAIA vers le modèle IF n'a été réalisé que pour montrer la faisabilité de l'automatisation. Dans la version actuelle, il ne permet d'effectuer une traduction que lorsque le modèle contient un seul composant d'adaptation. Enfin, la réalisation de transformation de modèle est une préoccupation importante récemment adressée par les travaux autour de l'ingénierie dirigée par les modèles ; ainsi de nombreux langages et approches de transformation de modèles apparaissent [33]. La transformation réalisée ici pour passer d'une description du modèle en XML à sa description en IF ne manipule pas directement les concepts permettant de passer d'un modèle à un autre mais est codée "en dur" dans du code java.

La mise en place de l'outil nous a permis de s'assurer de la cohérence entre les modèles. De plus, ceci nous a fourni un point d'entrée vers l'ingénierie dirigée par les modèles. On peut donc imaginer lier l'approche avec des approches pour les transformations de modèles et ainsi créer un pont vers d'autres langages pour la vérification, l'exécution, etc.

La section suivante présente la méthode proposée pour la réalisation de système de contrôle de processus avec SAIA.

4.2 Processus de modélisation incrémentaux dans SAIA

Afin d'utiliser efficacement les différents modèles proposés par le style architectural (qui peuvent s'avérer complexes), une méthode est proposée. Cette méthode n'est pas la seule pouvant être utilisée mais permet de montrer un enchainement possible pour la définition des différents modèles. La méthode définit des processus de modélisation incrémentaux (au sens de [79]). Elle permet donc de décomposer l'activité de modélisation en étapes et ainsi d'aider l'utilisateur à se concentrer sur un seul aspect de la mise en place de son modèle.

Nous avons vu que dans SAIA, l'application de contrôle, basée sur la plateforme abstraite est indépendante de la plateforme réelle. Pour ne pas être influencée par une plateforme réelle spécifique, la méthode proposée commence par la modélisation de l'application de contrôle et de sa plateforme abstraite. Ensuite, la modélisation de la plateforme réelle puis du connecteur complexe sont présentées. Enfin, une méthode permettant l'établissement des contrats est proposée.

4.2.1 Développement du SAIM

Avant la première étape de modélisation, il est nécessaire de posséder les spécifications du système à réaliser. La première étape de la modélisation du SAIM consiste à extraire des spécifications les différentes *entrées* et *sorties* nécessaires à l'application pour atteindre ses objectifs de contrôle (noté I.1 sur la figure 4.2). Comme spécifié dans la section 3.2.3 traitant du SAIM, il est nécessaire que la spécification des *entrées* et des *sorties* soit indépendante de la façon de les acquérir ou de les réaliser. La description des *entrées* et des *sorties* n'est pour l'instant que purement fonctionnelle. En effet, la description de la QoS est, à cet instant, impossible puisque fortement liée à l'application de contrôle.

La seconde étape est donc la modélisation de l'application de contrôle. Le contrôle est basé sur la description de la plateforme abstraite. La modélisation du contrôle en lui même est extrait des spécifications (noté I.2 sur la figure 4.2). Comme expliqué dans la section 3.2.3, la modélisation de l'application de contrôle n'est pas détaillée dans SAIA.

La troisième étape de modélisation consiste à extraire des spécifications les besoins en qualité de contrôle (noté I.3 sur la figure 4.2). La qualité de contrôle est souvent exprimée à l'aide de contraintes de haut niveau dans les spécifications. On trouve par exemple la spécification d'une échéance de bout en bout, d'une consommation électrique ou, de contraintes plus spécifiques à

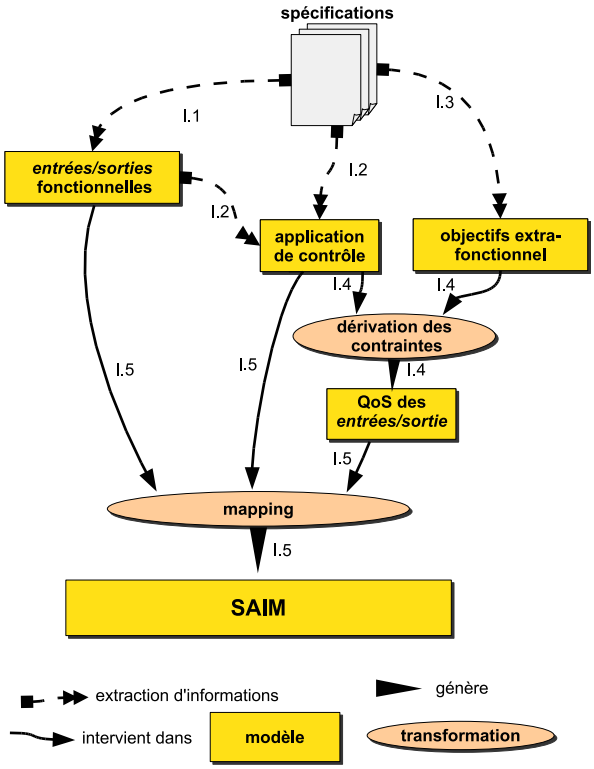


FIG. 4.2 – Les processus de modélisation incrémentaux pour la modélisation du SAIM

un contrôle particulier, comme par exemple une contrainte de déviation maximale lors d'un suivi de trajectoire.

Pour appliquer SAIA, il convient alors de dériver la qualité de contrôle de l'application afin d'obtenir la QoS correspondante sur chaque *entrée* et de chaque *sortie*. La qualité de contrôle n'est pas uniquement fonction de la qualité de service sur les *entrées* et les *sorties*. Elle est, entre autre fonction de la période d'activation de la loi de contrôle [11]. Cependant, dans SAIA la QoS est dérivée sur les *entrées* et les *sorties* après avoir fixé les autres paramètres influents. Cette étape, notée I.4 sur la figure 4.2, fournie la description de la QoS pour chacune des *entrées* et chacune des *sorties*.

La dernière étape (notée I.5 sur la figure 4.2) effectue la liaison entre la description fonctionnelle des *entrées* et des *sorties*, la description de la QoS et la modélisation du contrôle. Cette étape aboutit à un modèle SAIM complet.

4.2.2 Développement du SAM

Le développement du SAM consiste à spécifier la plateforme réelle. Il faut donc spécifier les différents flots d'informations circulant en entrée et en sortie de cette plateforme. Puisqu'il s'agit d'une plateforme réelle, cette étape est issue de l'analyse d'un système existant. Le SAIM peut être un ensemble composé de capteurs, d'actionneurs et de leur pilote ou bien un ensemble de

services d'un simulateur. Dans les deux cas, une description de la QoS doit être renseignée pour chacun des flots d'informations spécifiés.

Une fois le SAM modélisé, il est nécessaire de modéliser le connecteur complexe qui connecte le SAM et le SAIM, soit la description de l'ALM.

4.2.3 Développement de l'ALM

Nous proposons au travers de cette méthode la modélisation du connecteur complexe selon trois étapes. La première (notée II.1 sur le figure 4.3) spécifie la structure gros grains explicitant comment les pilotes sont impliqués dans l'acquisition ou la réalisation des *entrées* et des *sorties*. Autrement dit, des connexions logiques sont créées entre d'une part les interfaces des pilotes et celles des composants d'adaptation associés et d'autres part entre les interfaces des *entrées* et des *sorties* et celles des composants d'adaptation associés. Cette opération (notée III.1 sur la figure 4.4) nécessite la connaissance d'élément du SAM et du SAIM.

Les étapes suivantes spécifient la "glue" de chacun des sous connecteurs. La première en spécifie la configuration interne, c'est-à-dire quels sont les composants intervenants dans la "glue" et la manière dont ils sont connectés (notée II.2 sur la figure 4.3). Par exemple : Est-ce qu'un composant de type `Format` est nécessaire ; avant ou après un composant de type `Interpret` ? Une fois la configuration de la "glue" réalisée, il faut spécifier l'automate temporisé et/ou la fonction associés à chacun des composants de la "glue" (noté II.3 sur la figure 4.3). Durant cette étape il faut également spécifier la sémantique d'agrégation de la QoS s'il y a lieu.

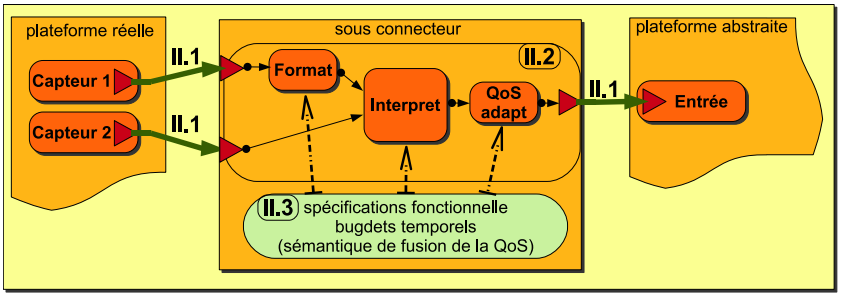


FIG. 4.3 – Les processus de modélisation incrémentaux pour la modélisation de l'ALM

Maintenant que l'ALM est spécifié, les trois modèles sont interconnectés (transformation noté III.2 sur la figure 4.4). En analogie au PSM (*Platform Specific Model*) défini dans MDA, nous appelons le modèle résultant de cette association le SASM (*Sensors Actuators Specific Model*). Ce modèle est fonctionnellement complet cependant nous avons vu dans la section 3.3.3 qu'il est nécessaire de réaliser une évaluation de la QoS et d'établir un contrat global de QoS pour s'assurer de la correction du système. C'est donc l'objet de la partie suivante.

4.2.4 Evaluation et contrat de QoS

L'étape d'évaluation de la QoS est réalisée en IF telle que cela est spécifié dans la section 3.3.3. Les informations nécessaires à la réalisation de l'évaluation sont issues du SASM. Puisque les évaluations sont effectuées sous connecteur par sous connecteur, un outil doit idéalement convertir le SASM en un ensemble de systèmes IF exécutables afin d'analyser la QoS modifiée en sortie de chaque sous connecteur (transformation notée III.3 sur la figure 4.4). La QoS modifiée ainsi obtenue peut être ajoutée au modèle SASM (transformation notée III.4 sur la figure 4.4).

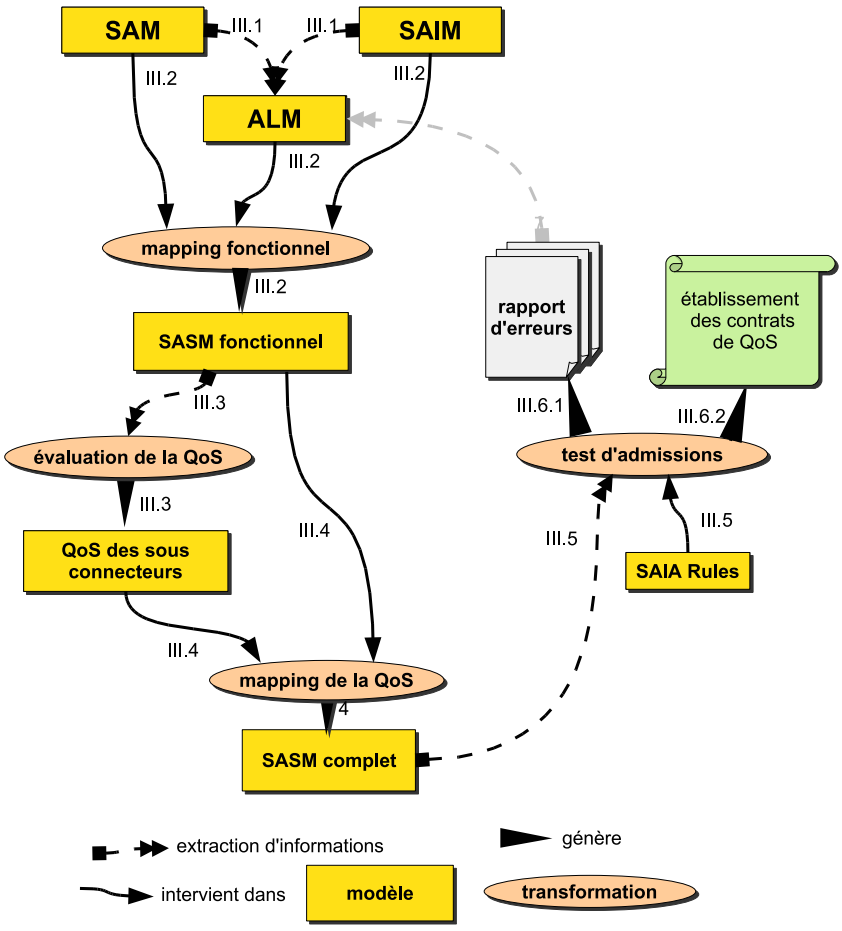


FIG. 4.4 – Les processus de modélisation incrémentaux pour la validation d'un système

Désormais, la QoS modifiée est connue en sortie de chaque sous connecteur, il reste à réaliser l'établissement des contrats afin de pouvoir établir le contrat de QoS global. L'établissement du contrat dépend de la technique utilisée pour décrire la QoS (constante, intervalle ou expression régulière). La relation de satisfaction est donc précisée dans le modèle noté *Rules* sur la figure 4.4. Lors du test d'admission (noté III.5 sur la figure 4.4), soit la QoS requise est satisfaite par la QoS fournie, un contrat est établi et on peut continuer l'établissement des contrats jusqu'à l'établissement du contrat global (III.6.2 sur la figure 4.4) ; soit un des contrats ne peut pas être établi, auquel cas un rapport d'erreur doit être généré et doit préciser quelle est la caractéristique de QoS qui ne peut pas être satisfaite. Il est alors possible d'adapter la QoS modifiée par le connecteur correspondant en ajoutant, par exemple, un composant `QoS_adapt` ou en modifiant

l'adaptation réalisée. Dans ce cas un nouveau processus de validation doit redémarrer à l'étape III.2 de la figure 4.4.

La partie suivante illustre cette méthode au travers d'un exemple simple.

4.3 Illustration de la méthode

Les concepts de SAIA ont été présentés dans les chapitres précédents. La mise en œuvre de SAIA ainsi que de la méthode proposée est illustrée dans cette partie par la réalisation partielle d'un robot explorateur dont la spécification sommaire est la suivante :

Le robot se déplace dans son environnement selon la consigne en trajectoire donnée par un opérateur. L'opérateur émet la trajectoire sous la forme d'un flot de données contenant une suite de positions absolues à atteindre. Le robot ne doit pas s'éloigner à plus de 0.08cm de la trajectoire désirée. L'environnement du robot a une superficie carrée de 400cm^2 la plus simple possible : elle ne contient pas d'obstacle, pas de trou, est plane et ne possède pas de pentes. Le robot peut alors se déplacer sur l'axe des X et des Y sans glissements. Enfin, à l'origine, le robot se trouve à la position à 100cm du bord gauche de la surface et à 10cm du bord inférieur.

4.3.1 Réalisation du SAIM

4.3.1.1 Étape 1 : extraction des *entrées* / *sorties*

La première étape consiste à extraire les *entrées* et les *sorties* des spécifications sommaires qui précèdent. Dans un premier temps l'objectif est de suivre une trajectoire. Celle-ci est exprimée par une *donnée* ayant le rôle *Consigne*. Elle est représentée par un couple [X ; Y] de nombres réels :

Consigne : `Position_a_atteindre`
type : [float ; float],
valeur minimum : [0 ; 0],
valeur maximum : [200 ; 200],
valeur initiale : [100 ;10],
unité : [cm ; cm].

On pourrait exprimer la consigne trajectoire en deux entrées distinctes : une pour X, l'autre pour Y. Dans ce cas la mise en place d'une synchronisation est nécessaire pour éviter que l'application puisse lire une position erronée lorsqu'une seule des données est mise à jour. SAIA ne donne aucune consigne sur ce niveau de modélisation.

Plusieurs mesures sont nécessaires pour effectuer le contrôle du robot. Le robot doit savoir où il est dans l'environnement afin d'atteindre la prochaine *Consigne*. De plus, le robot doit connaître son orientation afin de pouvoir s'orienter vers la prochaine position à atteindre. Ainsi

on distingue deux *données* possédant le rôle *Mesure*. La position actuelle [X ; Y] et l'orientation du robot par rapport au nord magnétique ; toutes deux représentées par des nombres réels :

Mesure : `Position_actuelle`
type : [float ; float],
valeur minimum : [0 ; 0],
valeur maximum : [200 ; 200],
valeur initiale : [100 ;10],
unité : [cm ; cm].

Mesure : `Orientation_actuelle`
type : float,
valeur minimum : -π,
valeur maximum : π,
valeur initiale : 0,
unité : radian.

Enfin, pour agir sur le processus robot, deux *sorties* de type `cmd` sont utilisées : une commande en vitesse notée `avancer_a` et une commande en rotation, notée `pivoter_de`, toutes deux représentées par des nombres réels :

Commande : `pivoter_de(param)`
type : float,
valeur minimum : -π,
valeur maximum : π,
valeur initiale : 0,
unité : radian.

Commande : `avancer_a(param)`
type : float,
valeur minimum : 0,
valeur maximum : 0.2,
valeur initiale : 0,
unité : cm/s.

Cet ensemble d'*entrées* et de *sorties* spécifie la plateforme abstraite nécessaire à la réalisation de l'application de contrôle du robot. Encore une fois, des différences peuvent apparaître dans la définition de la plateforme abstraite pour une même application de contrôle ; cependant la plateforme se doit de préciser l'ensemble des flots d'informations entre le monde extérieur et le système de manière indépendante d'une technologie spécifique. La modélisation fonctionnelle obtenue dans l'outil est présentée figure 4.5.

4.3.1.2 Étape 2 : Réalisation de l'application de contrôle

L'application de contrôle pour cet exemple a été réalisée grâce à l'outil Matlab/Simulink [82]. Nous avons vu que l'application de contrôle est basée sur la plateforme abstraite. En complément

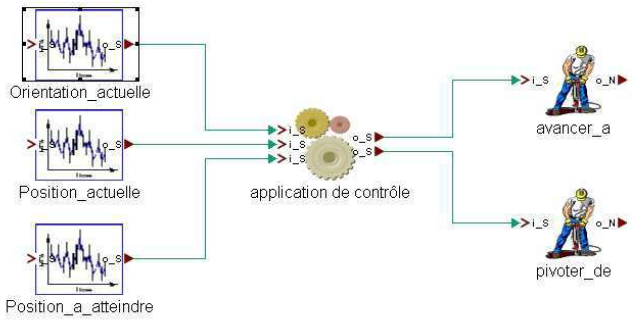


FIG. 4.5 – Modèle SAIM d'un robot explorateur simple dans l'outil SAIA

de la modélisation du monde extérieur, la plateforme abstraite est la première chose à modéliser dans Simulink. Ensuite, une application de contrôle simple a été réalisée : la position et l'orientation actuelle sont prises en compte périodiquement, le robot s'oriente vers la position à atteindre et avance. La vitesse, quand à elle est proportionnelle à la distance à parcourir entre la position actuelle et celle à atteindre au moment de la prise en compte de la consigne. Plus de détails sur la modélisation Simulink de cet exemple sont fournis dans [34].

4.3.1.3 Étape 3 : Extraction de la qualité de contrôle à respecter

La qualité de contrôle est, dans cet exemple, à la fois simple et explicitement précisée : "*Le robot ne doit pas s'éloigner à plus de 0.08cm de la trajectoire désirée*". La qualité de contrôle est, dans certains cas, exprimée de manière moins directe. Par exemple, dans le cas du robot, elle pourrait être exprimée par le fait que la trajectoire à suivre par le robot peut, dans le pire des cas, passer à 0,1cm d'un trou. Elle doit cependant toujours être exprimée.

4.3.1.4 Étape 4 : Dérivation de la qualité de contrôle en QoS

Plusieurs paramètres influent sur la qualité de contrôle du système. Nous fixons donc dans un premier temps tous les paramètres non relatifs à la plateforme abstraite. Dans notre cas, cela se résume à la période d'activation de l'application de contrôle, fixé à 110ms. Pour un système plus complexe, ces paramètres sont déterminés par des approches métiers, souvent issues du domaine de l'automatique.

Afin de présenter clairement l'impact de la QoS de la plateforme abstraite, nous fixons tous les paramètres de QoS de la plateforme abstraite sauf un : la loi de retard de l'*entrée* `Orientation_actuelle`. Ainsi, chacune des lois d'arrivée des autres *entrées* sont figées à 20ms et leur loi de retard à 0s. Pour les *sorties*, le retard est figé à 432ms.

De nombreuses simulations sont nécessaires pour déterminer la loi de retard sur l'`Orientation_actuelle` du robot afin que la qualité de contrôle soit respectée. La première simulation

considère un retard nul sur l'*entrée* `Orientation_actuelle`. Dans ce cas, la figure 4.6 montre l'erreur entre la trajectoire servant de *Consigne* et la trajectoire effectuée par le robot. Cette erreur est de 0.0713cm. L'erreur malgré la loi de retard nulle est due à la période d'activation de l'application de contrôle, à la loi d'arrivée des différentes *entrées* et à la loi de retard des *sorties*.

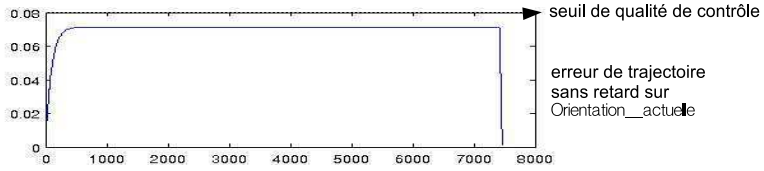


FIG. 4.6 – Erreur de trajectoire pour une loi de retard nulle

Un ensemble de simulations pour différentes valeurs de la loi de retard sur l'`Orientation_actuelle` du robot sont alors lancées dans le but de déterminer le langage le plus permissif possible assurant la correction du système. Dans le cas du robot, après une cinquantaine de simulation, nous avons abouti au langage suivant :

Loi de retard requise sur `Orientation_actuelle`	$[[0^{\omega} - 0.049^{\omega}] \mid [0 - 0.01]\{3,\}.[0.01 - 0.09]\{,4\}]^{\omega}$

Les valeurs de l'expression sont exprimées en seconde.

Cette expression exprime que, si le retard est constant il ne doit pas dépasser 49ms sans quoi, non seulement la qualité de contrôle n'est pas respectée mais, en plus, le robot n'arrive pas à suivre la trajectoire désirée (le système devient instable). Lorsque le retard est variable, le retard peut être compris entre 0 et 10ms et il peut dépasser ces valeurs un maximum de 4 fois tous les 3 retards entre 0 et 10ms. La valeur de l'excès doit alors être inférieure à 90ms. Ce langage n'est pas le plus permissif, il permet uniquement de montrer qu'une spécification de la QoS sous la forme d'une expression régulière plutôt que d'un intervalle permet d'obtenir plus de souplesse lors du déploiement sur une plateforme réelle tout en gardant une même qualité de contrôle. Ainsi, pour illustration sur la figure 4.7, le retard sur l'*entrée* `Orientation_actuelle` du robot forme un mot compris dans le langage de la loi de retard requise ; le système respecte la qualité de contrôle.

A contrario, sur la figure 4.8, la simulation a été effectuée avec un mot du retard de l'orientation actuelle légèrement différent de ceux définis comme acceptable par l'expression régulière de la QoS requise. En effet, au lieu d'avoir un retard compris entre 0 et 10ms lorsque l'on n'a pas d'excès, le début du mot considéré (figure 4.8) a un retard compris entre 0 et 30ms. On voit alors que l'erreur sur la trajectoire dépasse la qualité de contrôle désirée, représentée par une ligne pointillée.

4.3.1.5 Étape 5 : Finalisation du SAIM

Cette étape consiste uniquement à injecter dans l'outil de modélisation développé pour SAIA la QoS de la plateforme abstraite obtenue à l'aide des simulations présentées dans la section

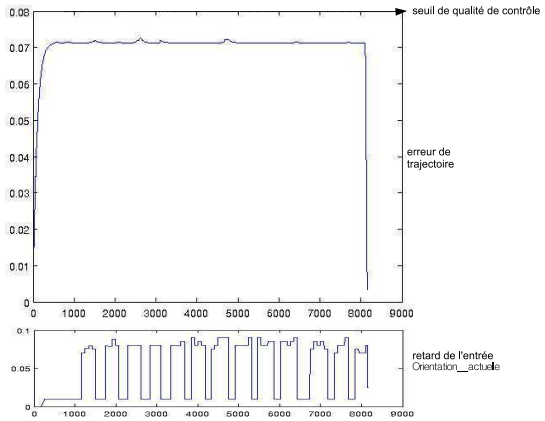


FIG. 4.7 – Erreur de trajectoire pour une loi de retard respectant le langage de retard requis

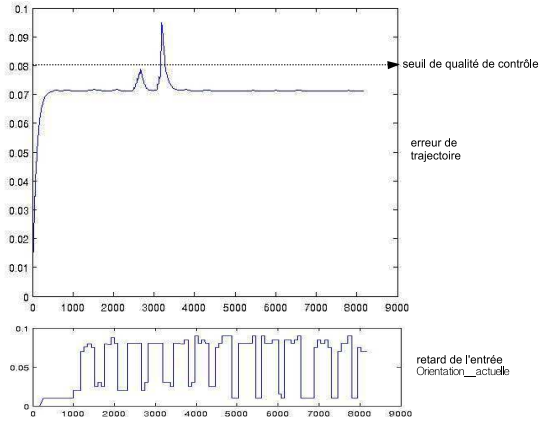


FIG. 4.8 – Erreur de trajectoire pour une loi de retard ne respectant pas le langage de retard requis

précédente.

4.3.2 Réalisation du SAM

Comme souvent, plusieurs SAM sont envisageables afin d'acquérir ou de réaliser les *entrées/sorties* spécifiées dans le SAIM. Par exemple, l'*entrée* `Position_actuelle` peut être fournie par un GPS (*Global Positioning System*) aussi bien que par des points de repère dans l'environnement. De la même façon, l'*entrée* `Position_a_atteindre` peut être fournie par la manipulation d'un joystick, par le suivi d'une ligne sur le sol, etc.

Il n'est pas rare que la plateforme réelle en termes de pilotes d'acquisition et de réalisation soit liée à une infrastructure physique particulière. Dans l'exemple choisi, l'infrastructure physique du robot est composée de deux roues motrices indépendantes de diamètre `d` et diamétralement opposées par un axe de longueur `d` sur lesquelles vient se greffer le "corps" du robot (cf. figure 4.9).

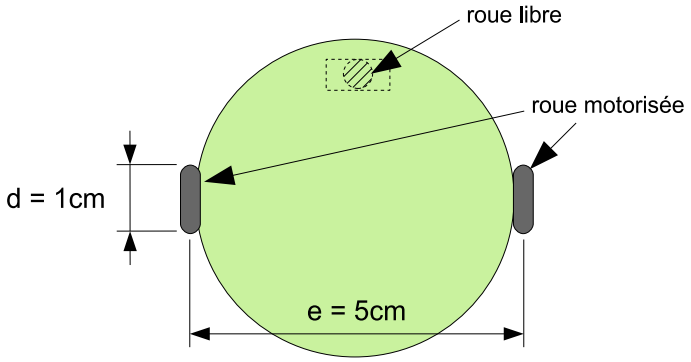


FIG. 4.9 – Infrastructure physique simplifiée liée à la plateforme réelle

La plateforme réelle est composée de quatre pilotes d'acquisition de données : la vitesse roue droite, la vitesse roue gauche, un GPS fournissant la position en coordonnées cartésienne et un pilote qui génère la position à atteindre en coordonnées polaires.

- `vitesse_roue_droite / vitesse_roue_gauche`
  type : float,
  valeur minimum : -30,
  valeur maximum : 30,
  valeur initiale : 0,
  unité : tour/s.
- `GPS`
  type : [float ; float],
  valeur minimum : [-100 ; -100],
  valeur maximum : [100 ; 100],
  valeur initiale : [100 ; 10],
  unité : [cm ; cm].
- `générateur_position`
  type : (float ; float),
  valeur minimum : (0 ; -π),
  valeur maximum : (0.1 ; π),
  valeur initiale : (100 ; 10),
  unité : [cm ; radian].

La plateforme réelle comporte également deux pilotes de réalisation de commandes : un pilote du moteur lié à la roue droite et un moteur lié à la roue gauche du robot.

- `roue_droite_avancer(param) / roue_gauche_avancer(param)`
  type : float,
  valeur minimum : 0,
  valeur maximum : 0.2,
  valeur initiale : 0,
  unité : cm/s.

La modélisation fonctionnelle réalisée dans l'outil SAIA est présentée sur la figure 4.10

L'étape d'analyse de la QoS de chacun des pilotes n'est pas détaillée ici. Cependant, l'analyse des deux capteurs d'acquisition de données `vitesse_roue_droite` et `vitesse_roue_gauche` donne une loi de retard conforme à l'expression régulière suivante :

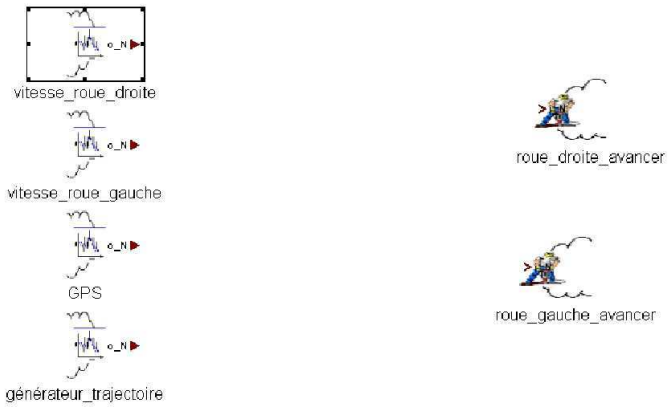


FIG. 4.10 – Modèle SAM d'un robot explorateur simple dans l'outil SAIA

$$[[0.005]\{5,\}.[0.01-0.06]\{,1\}]^{\omega}$$

Les valeurs de l'expression sont exprimées en seconde.

Textuellement, chacun des pilotes fournit des occurrences avec un retard de 0.005 seconde mais pour une occurrence sur 6, le retard peut être compris entre 0.01 et 0.06 seconde.

4.3.3 Réalisation de l'ALM

4.3.3.1 Étape 1 : structure des sous connecteurs

Cette étape permet de spécifier les pilotes qui entrent dans l'acquisition ou la réalisation de chaque *entrées* et de chaque *sorties*. La structure du système est donnée sur la figure 4.11.

On voit que le pilote du `GPS` permet l'acquisition de l'*entrée* `Position_actuelle`, les pilotes `vitesse_roue_droite` et `vitesse_roue_gauche` fournissent l'*entrée* `Orientation_actuelle`. La vitesse fournie par les pilotes d'acquisition est exprimée en tour par seconde. Afin de pouvoir calculer l'orientation du robot, il est nécessaire de connaître d'une part le diamètre des roues du robot (pour calculer la vitesse sur la roue droite et sur la roue gauche en centimètre par seconde) et d'autre part de connaître la taille de l'axe entre les deux roues (pour en déduire la rotation du robot et de fait, son orientation). Ces deux informations ne varient pas lors de l'exécution et sont donc renseignées à l'aide de deux constantes notées `d` pour le diamètre des roues et `e` pour l'écart entre les deux roues. L'*entrée* `Position_a_atteindre`, ayant le rôle de *Consigne* est acquise par le pilote `générateur_position`. Enfin les deux *sorties* `avancer_a` et `pivoter_de` sont réalisées toutes deux par les pilotes `roue_droite_avancer` et `roue_gauche_avancer`.

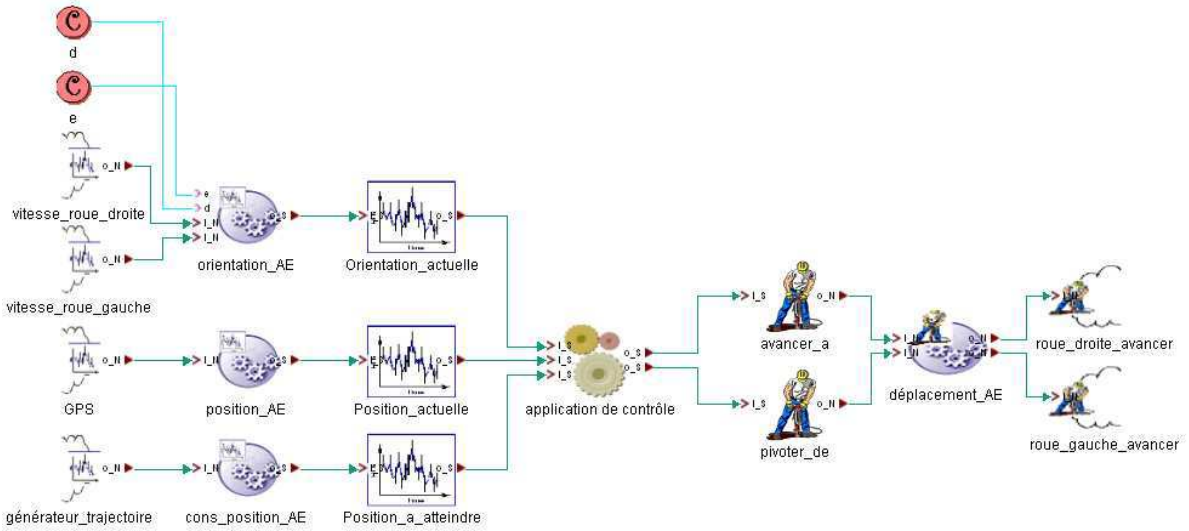


FIG. 4.11 – Modèle SASM d'un robot explorateur simple dans l'outil SAIA

4.3.3.2 Étape 2 : structure de la "glue" des sous connecteurs

Cette étape et la suivante ne sont détaillées que pour le sous connecteur en charge de l'adaptation entre les pilotes `vitesse_roue_droite` et `vitesse_roue_gauche` afin qu'ils fournissent l'*entrée* `Orientation_actuelle`. Ce sous connecteur utilise les deux constantes décrites dans la section précédente. En effet, les pilotes de vitesse des roues fournissent un flot de données exprimées en tour/s, le diamètre `d` est utilisé par les deux composants de type `Format` afin de traduire cette vitesse en cm/s. De plus, l'écart `e` est utilisé par le composant de type `Interpret` afin de calculer la rotation du robot et en déduire son orientation. On remarque que le composant de type `Interpret` est également en charge de la fusion des deux flots de données en provenance des composants de type `Format` (cf. figure 4.12).

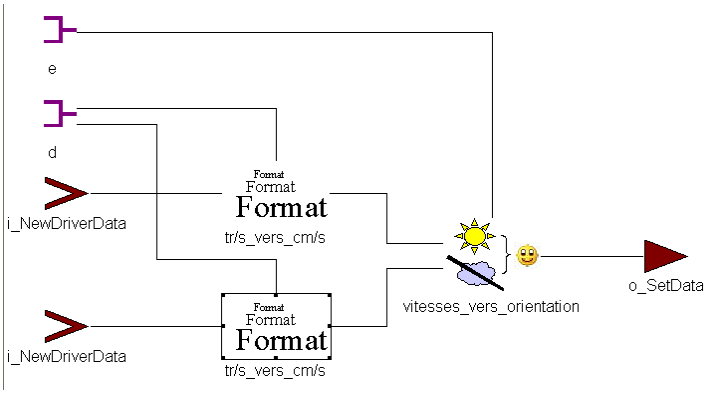


FIG. 4.12 – Structure de la "glue" du sous connecteur réalisant l'acquisition de `Orientation_actuelle`

4.3.3.3 Étape 3 : spécification des composants de la "glue"

Durant cette étape, chacun des trois composants spécifiant la "glue" présentée précédemment est spécifié. Pour les composants de type `Format`, il est nécessaire de spécifier la fonction associée. Cette fonction est abstraite par sa durée d'exécution dont le budget temporel est fixé à 10ms. Pour la spécification du composant de type `Interpret`, deux choix différents peuvent être faits. Soit, on choisit un automate prédéfini dans la boîte à outils fournie par l'outil SAIA, soit on construit un automate particulier. Un automate prédéfini a été choisi, il génère une occurrence à chaque fois qu'il reçoit une occurrence de chacun de ses flots d'entrée. Une fois l'automate choisi, il reste à spécifier la fonction en charge de la fusion des deux occurrences collectées. Cette fonction est également abstraite par ses informations extra-fonctionnelles. Ainsi, deux budgets temporels sont définis : le BCET, égal à 30 ms et le WCET, égal à 40ms.

Une fois les étapes 2 et 3 réalisées (cf. section 4.2.3) pour chacun des sous connecteurs, on obtient un système complet du point de vue fonctionnel. La QoS en sortie de chaque sous connecteur doit maintenant être évaluée afin de permettre l'établissement des contrats de QoS.

4.3.4 Réalisation de l'évaluation de la QoS

Comme pour les deux étapes précédentes, ce chapitre ne détaille que l'ensemble comprenant les deux pilotes de vitesse des roues ainsi que le sous connecteur et l'*entrée* associés. Nous possédons tous les éléments permettant de connaître la QoS modifiée en sortie de ce sous connecteur :

- l'expression régulière exprimant la QoS requise par l'*entrée* sur la loi de retard ;
- l'expression régulière exprimant le retard fourni par chacun des pilotes ;
- le comportement du sous connecteur.

Malgré la simplicité apparente de l'exemple, il est difficile de prédire si le contrat peut être établi ou non. Cette partie du système sous étude est exécutée par la suite d'outils IF. Suite à l'exécution, après avoir vérifié que le LTS obtenu est un graphe cyclique sans état bloquant, après avoir caché ses transitions non pertinentes et après avoir effectué les réductions, on obtient le LTS exprimant la QoS modifiée en sortie du sous connecteur (cf. section 3.3.3).

La prochaine étape consiste alors à vérifier si le contrat peut, ou ne peut pas, être établi.

4.3.5 Réalisation de l'établissement du contrat de QoS

Un contrat peut être établi si l'équivalence de sous langage entre la QoS fournie et la QoS requise est vérifiée. Ceci est réalisé grâce à l'outil "bisimulator" de la suite CADP [46]. Afin de vérifier si la relation de sous langage existe, nous utilisons la relation d'équivalence de trace implémentée dans CADP. Pour le sous connecteur présenté précédemment, l'équivalence de trace est vérifiée. La relation QoS_requise "*satisfies*" QoS_fournie est donc vrai et le contrat peut être établi. Cependant, il faut remarquer que si l'on change le budget temporel du WCET de la fonction comprise dans le composant de type `Interpret` du sous connecteur de 40ms à 50ms, la relation de sous langage n'est plus satisfaite. Dans ce cas l'outil bisimulator donne le diagnostic suivant :

```
does providedQoS.aut trace_equivalent_to requiredQoS.aut ?
bcg_open : using "/usr/local/cadp/bin.iX86/bisimulator.a"
bcg_open : running "bisimulator -smaller -diag -trace requiredQoS.bcg" for "./providedQoS.bcg"
FALSE

** diagnostic sequence found at depth 5
<start state>
"DLocc= p1=11 "
"DLocc= p1=11 "
"DLocc= p1=11 "
"DLocc= p1=11 "
"Absent in requiredQoS.bcg : DLocc= p1=11 "
<goal state>
```

Ce diagnostic indique qu'une violation est détectée sur $QoSocc_5$. En effet, l'expression régulière décrivant les mots de la QoS requise (cf. section 4.3.1.4) spécifie que le nombre d'occurrence de retard peut excéder 10ms jusqu'à hauteur de 90ms mais seulement un maximum de 4 fois consécutives toutes les 7 occurrences. Ainsi l'outil diagnostique qu'une suite de 5 occurrences consécutives dépassant 10ms (dans ce cas égale à 11ms) ne fait pas parti du langage de la QoS requise. Le contrat ne peut donc pas être établi

4.3.6 Conclusion sur l'illustration de la méthode

L'exemple présenté ci-avant permet de montrer l'enchaînement des étapes de la méthode proposée. Il permet également de montrer l'intérêt d'une spécification précise de la QoS sur les *entrées* et les *sorties* afin d'atteindre une certaine qualité de contrôle par l'application. Ce document ne traite pas la manière de dériver les contraintes de la qualité de fonctionnement en contrainte de QoS sur les *entrées* et les *sorties*.

Une fois la QoS sur la plateforme abstraite (les *entrées* et les *sorties*) spécifiée et la QoS modifiée par un sous connecteur évaluée, la relation de sous langage doit être vérifiée pour établir un contrat. Dans le cas où la relation de sous langage n'est pas respectée, l'outil vérifiant la relation permet d'indiquer la première des occurrences de QoS fautive. Il est à noter que pour la connaissance de l'occurrence de QoS fautive n'est pas suffisante pour indiquer la modification à réaliser dans la "glue" pour pouvoir établir le contrat de QoS.

Nous montrons maintenant une utilisation alternative de SAIA pour la mise au point temporelle du système.

4.4 SAIA pour la mise au point de systèmes

Nous avons vu que l'utilisation de l'architecture et des outils IF associés permet d'évaluer la QoS résultant d'une configuration de composants SAIA. Cette évaluation peut être fournie de manière précise à l'aide d'expressions régulières mais aussi de manière bornée à l'aide d'intervalles. Il est donc possible d'évaluer, par exemple, le retard maximum de la donnée en sortie d'un connecteur pour une configuration donnée. Les outils d'évaluation peuvent alors servir d'aide

à la mise au point d'un système. Il est par exemple possible d'effectuer une série d'évaluation en faisant varier un ou plusieurs paramètre(s) temporel(s) d'un système afin de choisir une configuration respectant la propriété recherchée.

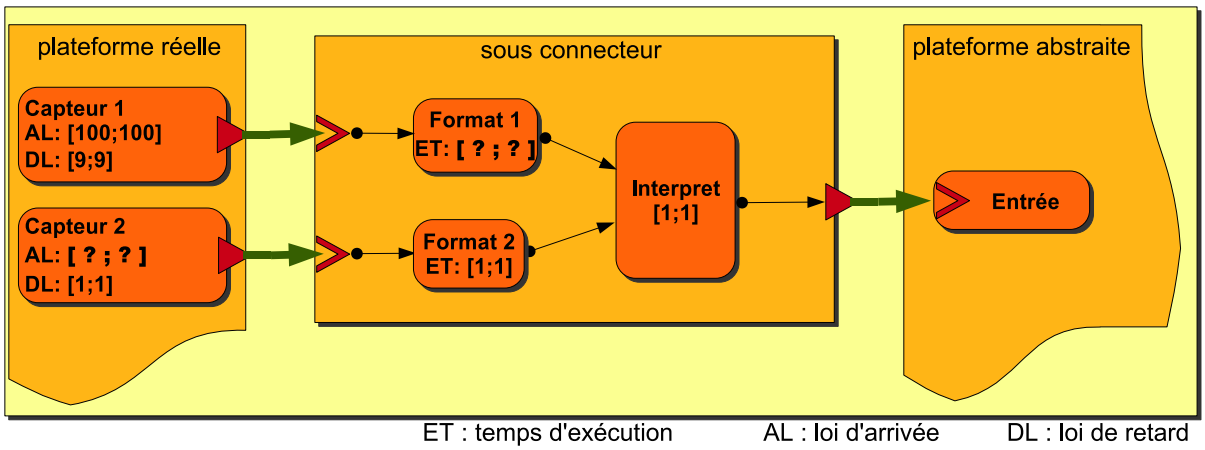


FIG. 4.13 – Exemple de variation de deux paramètres temporels dans une configuration

Pour illustrer cette utilisation de SAIA, nous considérons un système représentant un banc de test moteur, nécessitant la connaissance de la masse volumique de l'air entrant dans le moteur. La masse volumique d'un gaz est calculée en fonction de sa température et de sa pression. Afin d'obtenir une bonne estimation de la masse volumique du gaz il est nécessaire de posséder des valeurs de la température et de la pression ayant une corrélation temporelle faible. Dans ce système nous considérons la construction de l'*entrée* Masse volumique. Pour ce faire, un premier pilote d'acquisition noté `Capteur 1` fournit un flot de données représentatif de l'évolution de la pression de manière périodique toutes les 100ms. Lorsque le pilote d'acquisition fournit une occurrence, celle-ci possède un retard initial de 9ms. Un deuxième pilote d'acquisition (noté `Capteur 2`) fournit un flot de données représentant l'évolution de la température. Le flot de donnée est périodique et sa période peut être ajustée entre 10 et 30ms. Quelque soit la période du pilote d'acquisition, le retard initial de ses occurrences est de 1ms. Avant d'être fusionnées, les flots de données subissent une étape de formatage. Le temps d'exécution de la fonction de formatage du flot fournissant la pression peut être ajustée entre 1 et 19ms alors que la fonction de formatage du flot de température est de 1ms. Sachant que la fonction de fusion a un temps d'exécution de 1ms, on désire choisir une configuration garantissant une corrélation temporelle inférieure ou égale à 15ms en ajustant les paramètres du système (la période de `Capteur 2` et le temps d'exécution de `Format 1`).

Afin de répondre à ce problème, il est possible de lancer une série d'évaluation de la QoS pour chaque valeur de la loi d'arrivée de `Capteur 2` et pour chaque temps d'exécution de `Format 1`.

Pour lancer la série d'évaluation il est nécessaire de contraindre les hypothèses concernant la synchronisation entre les pilotes d'acquisition. Typiquement, lors d'une analyse de type RMA, aucune hypothèse n'est faite sur la synchronisation des pilotes d'acquisition du système. Cette analyse donne alors une borne maximale souvent pessimiste. La représentation IF du modèle permet d'affiner ce paramètre en fournissant des bornes de minimales et maximales de synchronisation.

Une fois les contraintes de synchronisation spécifiées, il est possible de lancer la série d'évaluation de la QoS et ainsi de définir l'ensemble des couples de la loi d'arrivée et du temps d'exécution permettant d'obtenir une corrélation temporelle maximum inférieure à 15ms. Afin d'illustrer graphiquement les résultats de cette approche, nous avons réalisé deux séries d'évaluation. La première fait l'hypothèse que les pilotes sont synchronisés. On voit alors l'évolution de la corrélation temporelle maximale (axe `Z`) selon la loi d'arrivée de `Capteur 2` et de `Format 1`. Sur les figures 4.14 et 4.15, les configurations permettant d'obtenir une corrélation temporelle inférieure ou égale à 15 sont colorées en orange. Sur la figure 4.14, on remarque une évolution non intuitive de la corrélation temporelle. On remarque également les points de synchronisation intéressants du système (période de `Capteur 2` égale à 20 ou 25ms). La deuxième série d'évaluation fait l'hypothèse que les pilotes d'acquisition sont, dans le pire des cas, déphasés de 30ms. Puisque l'on regarde l'évolution de la corrélation temporelle maximale, on remarque, sur la figure 4.15, que certains phénomènes de synchronisation sont atténués. Les phénomènes de synchronisation intervenant pour une période de `Capteur 2` égale à 20 ou 25ms sont cependant toujours présents.

Sur les deux représentations graphiques, on remarque qu'il est possible de choisir un couple acceptable possédant un temps d'exécution de `Format 1` et une période de `Capteur 2` élevés. Ceci permet, entre autre de ne pas surcharger le système en ne considérant que les cas les plus pessimistes.

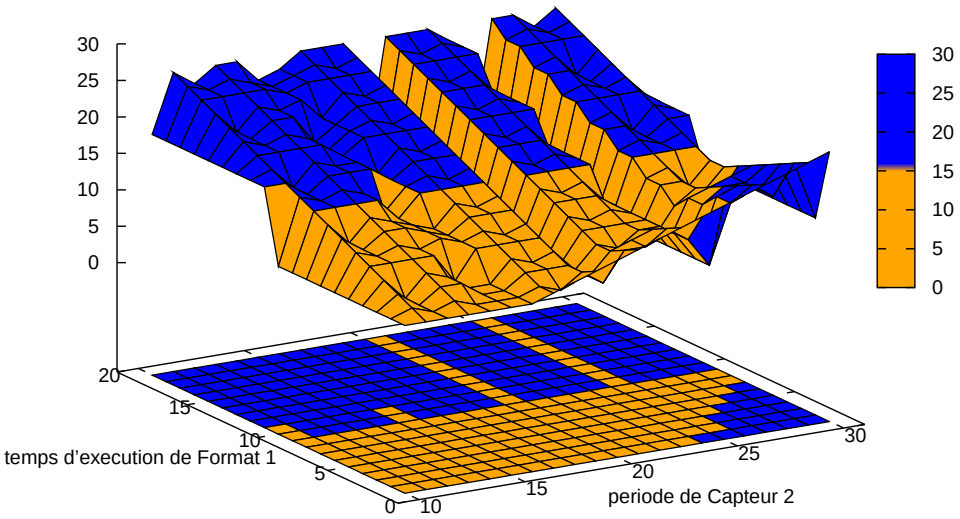


Fig. 4.14 – Corrélation temporelle maximale de la donnée agrégée lorsque les pilotes des capteurs sont synchronisés

L'exemple utilisé ici utilise des spécifications de la QoS sous forme d'intervalle. De plus la valeur extraite de l'évaluation est une borne maximale de la corrélation temporelle. Enfin, nous ne faisons varier que deux paramètres dans le système. Ceci a pour de but de pouvoir représenter graphiquement les résultats. Cependant, il est également possible d'utiliser cette approche avec des expressions régulières et chacune des relations de satisfaction présentées dans la section 3.3.4.

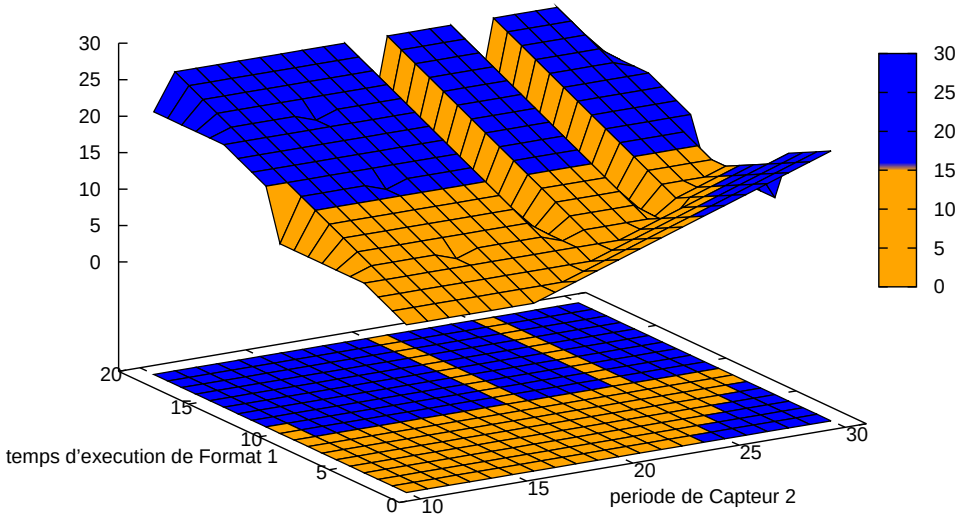


FIG. 4.15 – Corrélation temporelle maximale de la donnée agrégée lorsque les pilotes des capteurs sont désynchronisés au maximum de 30ms

Il est également possible de faire varier plus de deux paramètres.

Maintenant que les utilisations possibles de SAIA ont été décrites et illustrées, les sections suivantes présentent la mise en œuvre de SAIA, réalisée pour la modélisation de deux cas d'étude plus conséquent : *maRTian Task*[2] et *CiberMouse* [3]. Ce sont tous deux des robots explorateur proposés lors de concours internationaux.

4.5 Concours n° 1 : *The maRTian Task*

Le concours "maRTian Task" a été proposé en temps d'évènement satellite de la conférence *Real Time System Symposium* en décembre 2005. Le but est le développement du contrôle d'un robot explorateur devant aller de leur position d'origine à un point B dans un environnement a priori inconnu. Le robot a une vue limitée du monde et doit impérativement éviter les trous afin de rester en vie. Ce concours a donné lieu à une compétition durant laquelle le but était d'arriver le premier au point B. La plateforme d'exécution du robot n'était pas imposée mais sa plateforme de communication avec l'environnement est basée sur un simulateur fourni par les organisateurs du concours. L'approche SAIA a donc été employée d'une part pour réaliser une mise en œuvre réelle de l'approche et d'autre part pour montrer l'intérêt d'une telle approche lors d'un développement basé sur un simulateur.

[2] http ://www.ertos.nicta.com.au/events/05/rtss_competition.pml
[3] http ://www.ieeta.pt/∼lau/web_ciberRTSS/info.htm

Après une présentation du simulateur fourni pour le concours, cette section aborde succinctement les problèmes qui apparaissent lorsqu'une application de contrôle est développée sur un simulateur avant d'être déployée sur une plateforme réelle. Ensuite, quelques solutions liées à l'utilisation de SAIA sont proposées pour répondre à ces problèmes. Enfin, les modèles ainsi que leur mise en œuvre sont rapidement présentés.

4.5.1 Le simulateur associé à *maRTian Task*

Le simulateur a été développé par *the Institute for Real-Time Computer Systems* de Munich et modifié à l'occasion du concours par l'équipe *Embedded and Real-time Operating Systems* de l'université de Sydney (National ICT Australia Ltd.). Il fournit des flots de données sur l'environnement et consomme des flots de commandes et de commandes événementielles pour contrôler le procédé. Plus précisément, le simulateur possède 4 pilotes d'acquisition de données permettant de connaître l'état du monde extérieur :

- pilote fournissant l'état des bumpers du robot (collision),
- pilote fournissant l'état des sonars du robot (présence d'un trou),
- pilote fournissant l'état du robot (en marche, mort, etc),
- pilote fournissant la position et l'orientation du robot.

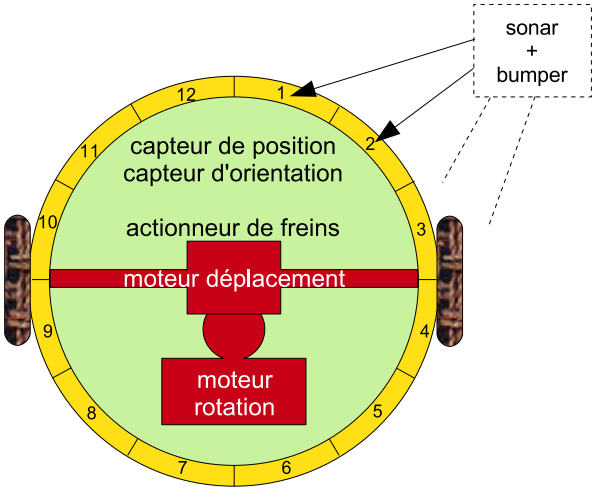


FIG. 4.16 – Présentation de l'infrastructure physique virtuelle de *maRTian Task*

Le simulateur comprend également trois pilotes de réalisations permettant de contrôler le procédé, les deux premiers sont des pilotes de réalisation de commandes et le dernier un pilote de réalisation de commandes événementielles :

- pilote réalisant la rotation du robot,
- pilote réalisant le déplacement du robot,
- pilote réalisant l'arrêt du robot.

Chacun des pilotes est fourni avec une spécification de sa QoS en termes de loi d'arrivée minimale et maximale. De plus le simulateur introduit des erreurs de lecture des capteurs sous la forme de probabilité. Ainsi, des faux négatifs et des faux positifs peuvent apparaître. Une représentation simplifiée de l'infrastructure virtuelle du robot est donnée figure 4.16.

De cette description du simulateur, fournie pour le concours, découle le modèle SAM. Le modèle SAIM n'est pas détaillé ici. On remarque cependant sur le figure 4.17 que les *entrées* et les *sorties* identifiées dans la plateforme abstraite ne font pas de suspicions sur le SAM sous-jacent. De plus, la plateforme abstraite contient uniquement les *entrées* et les *sorties* nécessaires à la réalisation de la mission décrite pour le concours.

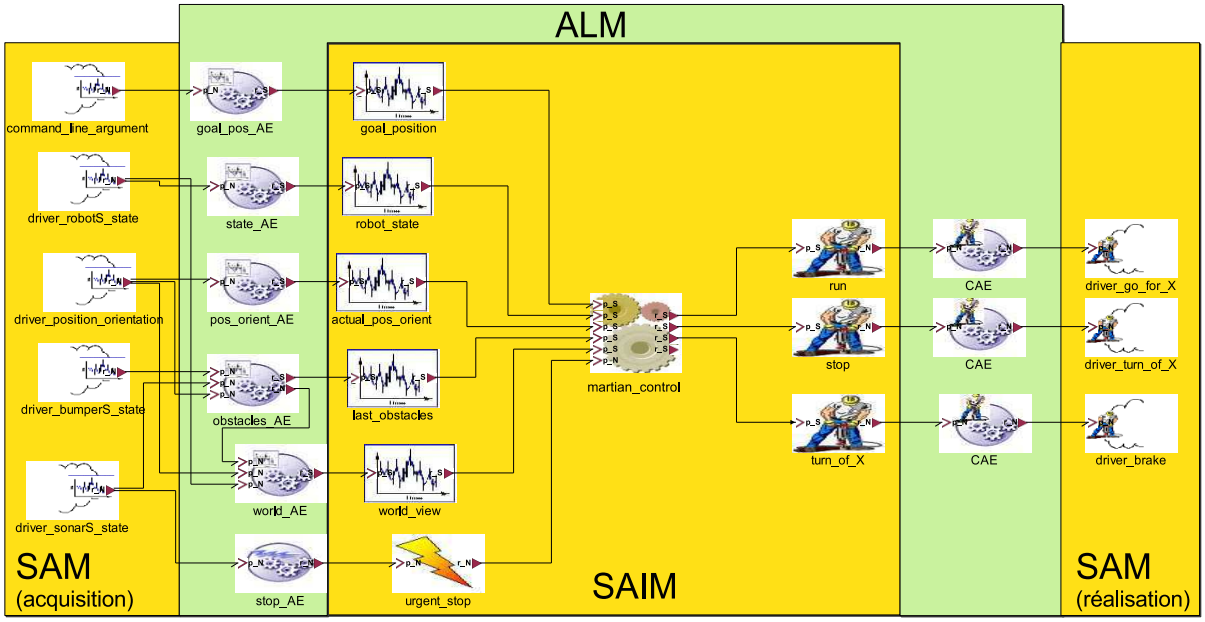


FIG. 4.17 – Présentation du modèle SASM obtenu pour *maRTian Task*

4.5.2 Déployer l'application de contrôle sur une cible réelle

L'utilisation d'un simulateur pour le développement d'une application de contrôle est une aide précieuse. Cependant des problèmes dus au manque de réalisme des simulateurs interviennent lors du déploiement de l'application de contrôle sur une cible réelle. Le manque de réalisme peut intervenir au niveau fonctionnel comme au niveau extra fonctionnel. Si ces déviances par rapport à une cible réelle ne sont pas maîtrisées, elles peuvent modifier, lors du déploiement, la structure et la qualité de contrôle du système (sa précision, voire sa stabilité).

Au niveau fonctionnel, les services fournis par le simulateur et une plateforme réelle peuvent être différents : il peut, par exemple, exister un pilote par sonar, un pilote pour la position différent de celui pour l'orientation, la position peut être donnée de manière relative par rapport au point de départ, etc. De la même façon, au niveau extra fonctionnel, des différences entre le simulateur et une cible réelle apparaissent. Par exemple, dans *maRTian Task*, la lecture des sonars est perturbée de manière probabiliste par l'apparition de faux positifs et de faux négatifs. Le manque de réalisme à ce niveau vient du fait que lorsqu'un faux négatif (ou un faux positif) apparaît, des lectures successives du même capteur donnent le même faux négatif (ou faux positif) tant que le robot n'a pas bougé (tourné, avancé ou reculé). Ce type de comportement est donc clairement différent de celui d'un capteur réel et peut impacter la validité de l'application de contrôle. Un autre manque de réalisme du simulateur est le fait que l'envoi d'une commande

d'arrêt au pilote de réalisation arrête le robot instantanément. Un vrai robot ne peut pas s'arrêter instantanément, il possède un minimum d'inertie, ce qui implique un retard avant l'arrêt du robot. Enfin, les retards des occurrences en provenance des capteurs sont nuls. Encore une fois ces valeurs manquent de réalisme.

4.5.3 SAIA et le développement basé sur un simulateur

Comme nous venons de le voir au travers du simulateur proposé pour le concours, il est difficile d'obtenir un simulateur réaliste. Il est alors difficile de tester (voire de valider) une application de contrôle sur un simulateur en espérant préserver ce comportement lors du déploiement sur une plateforme réelle. Afin de contourner ce problème, il est possible de modifier, au sein du connecteur complexe, la QoS fournie/requise par la le simulateur et ainsi s'approcher d'une QoS réaliste. Ceci peut par exemple se faire par l'ajout de retard, par un filtrage de certaines occurrences, etc. En particulier dans notre cas, il est intéressant d'augmenter le retard sur l'arrivée des occurrences de sonars et sur l'arrêt du robot afin de trouver, a priori, les lois de retard permettant au robot de ne pas tomber dans les trous une fois le déploiement effectué. Il peut également être intéressant de filtrer certaines occurrences du flot de données en provenance du GPS afin de trouver, encore une fois, les limites permettant au robot d'effectuer sa mission.

Ainsi, de la même façon qu'il est possible de dériver les contraintes de qualité de fonctionnement en QoS sur la plateforme abstraite à l'aide d'un simulateur de la plateforme abstraite sous Matlab, il est possible ici, de dériver les contraintes à l'aide d'un simulateur de la plateforme réelle et d'une modification de sa QoS par le connecteur complexe.

Maintenant que le simulateur et l'utilisation de SAIA pour le développement sur simulateur a été présenté, la section suivante présente les choix réalisés pour l'implémentation des différents modèles.

4.5.4 La mise en œuvre des modèles

La modélisation du système en accord avec les spécifications fournies par *maRTian Task* a permis d'aboutir au modèle SASM décrit par la figure 4.17

Une fois le système modélisé, il est nécessaire de l'implémenter dans un langage spécifique et d'établir un modèle de tâches [8]. Il existe différentes stratégies de mise en œuvre d'une configuration (cf. section 2.2.5). Notre premier choix fut d'exclure l'utilisation d'un intergiciel mais plutôt de compiler la configuration pour obtenir un code monolithique. Ceci est entre autres motivé, aux vues du concours, par l'inutilité de réifier les composants à l'exécution et la recherche d'un comportement temporel optimisé. Nous avons donc choisi de réaliser l'implémentation en C++. À un composant nous faisons correspondre une classe et un objet. Dans le but de se rapprocher de la sémantique opérationnelle de IF, la communication entre les objets passe par une zone de mémoire partagée permettant de représenter une file d'attente de type FIFO et de taille 1.

Pour atteindre les objectifs temps réel, il est nécessaire de fournir une gestion de la concurrence entre les objets. On trouve deux approches principales pour gérer cette concurrence : "*event-based*" and "*object-based*" [90]. Résumée rapidement, la stratégie *object-based* crée une tâche par objet alors que l'approche "*event-based*" crée une tâche par fil d'événement temps réel. La stratégie "*object-based*" est plus simple à mettre en œuvre mais augmente les sections critiques ainsi que l'utilisation du CPU. A contrario, la stratégie "*event-based*" permet de réduire les temps de blocages mais demande la mise en place explicite d'un modèle de tâches orthogonal aux objets ; ce fut la stratégie choisie. Le modèle de tâches est orthogonal à celui des objets puisque lié aux événements temps réel. Chaque action déclenchée par l'événement temps réel est exécutée dans une tâche spécifique. Dans *maRTian Task*, le seul événement déclenchant une tâche temps réel est la demande d'arrêt d'urgence (`urgent_stop`), déclenché sur la détection d'un trou par le pilote d'acquisition des sonars. Une tâche événementielle est donc créée et liée à l'élément déclencheur (l'activation d'un sonar). Les opérations des objets permettant d'exécuter l'arrêt du robot sont alors exécutées dans cette tâche. L'événement déclenchant potentiellement la tâche est issu de la lecture d'une détection d'un sonar dans le flot de données acquis par le pilote associé. Les pilotes d'acquisition ont tous la même loi d'arrivée périodique, les envois des occurrences qu'ils génèrent sont donc regroupés au sein d'une même tâche, déclenchée périodiquement. Enfin, le reste des activités (recherche de chemin, politique d'évitement) sont regroupé au sein d'une autre tâche, également déclenchée périodiquement. On distingue ainsi trois tâches :

- la tâche périodique réalisant la lecture des capteurs (en orange figure 4.18),
- la tâche sporadique réalisant l'activité urgente (en bleu figure 4.18),
- la tâche périodique regroupant les activités non urgentes (en jaune figure 4.18).

Le modèle de tâches est présenté sur la figure 4.18. On remarque la présence d'objets partagés entre les tâches (en vert figure 4.18). Ces objets sont protégés par un accès en exclusion mutuelle.

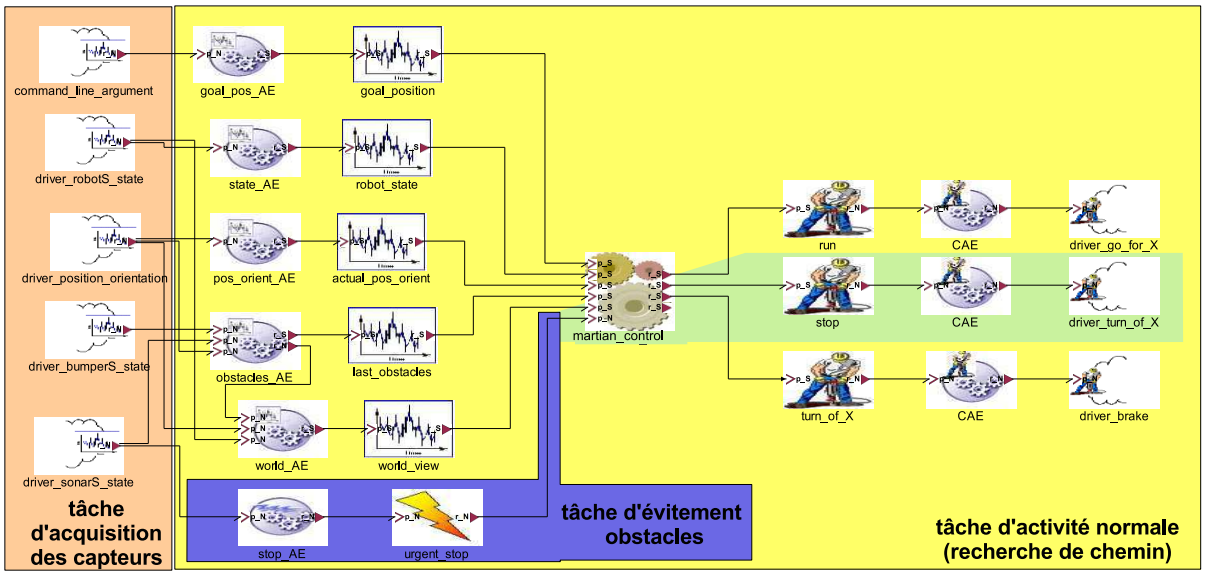


FIG. 4.18 – Présentation du modèle des tâches choisi pour *maRTian Task*

Cette mise en œuvre n'a pour but ni de choisir une implémentation pour SAIA ni de formaliser la mise en œuvre des modèles dans SAIA mais plutôt de s'assurer de la faisabilité des modèles à

l'exécution. De plus, cela nous a permis de nous rendre compte que la structuration gros grains et la séparation des aspects dépendant et indépendant de la plateforme facilite l'implémentation du système. En effet, une fois les modèles réalisés, chacun des composants peut être développé de manière indépendante des autres composants.

Si l'on désire s'assurer de la conformité de l'implémentation par rapport aux modèles présentés, des analyses supplémentaires doivent être réalisées afin de s'assurer que la sémantique opérationnelle de l'implémentation est conforme à celle des modèles (c.a.d. à celle de IF).

L'implémentation des modèles, nécessaires pour répondre aux exigences de ce concours, nous a permis de produire un code qui a rapporté une deuxième place lors de la compétition. Ceci illustre que génie logiciel et performances ne sont pas obligatoirement contradictoires. Maintenant que *maRTian Task* a été présenté, la réutilisation de son modèle SAIM dans le cadre du concours "*The CiberMouse Project*" est présentée dans la partie suivante.

4.6 Concours n° 2 : *The CiberMouse Project*

Le but de ce concours était, comme pour le concours numéro 1, le développement, basé sur un simulateur, du contrôle d'un robot explorateur. Le but du contrôle est quelque peu différent de celui de l'année précédente et le simulateur est, lui, complètement différent. Afin de remporter le concours, il est principalement demandé que le robot effectue sa mission plus rapidement que ses adversaires. Cependant, notre participation avait pour but d'évaluer la propension de SAIA à réutiliser un modèle SAIM sur des plateformes différentes.

Après une description de *CiberMouse*, les points communs et les différences entre les missions et les simulateurs des deux concours sont présentés. Ensuite, une deuxième puis une troisième partie présente respectivement les changements apportés au SAIM et la modélisation du connecteur complexe. Enfin, une dernière partie conclue sur une critique de l'approche.

4.6.1 *CiberMouse* versus *maRTian Task*

Durant le premier concours, le but de chaque robot est d'aller de leur position d'origine à un point B, tous deux connus a priori, sans tomber dans un trou. Dans le deuxième concours, la tâche des robots est d'aller de leur position d'origine à une surface cible signalée par un émetteur de lumière infrarouge et de retourner à leur position d'origine. Le score est d'autant meilleur que le robot effectue sa tâche rapidement et sans collision contre les murs ou les autres robots en jeu.

La position d'origine, la surface cible, les murs et les autres robots sont inconnus du robot à contrôler.

Dans les spécifications de *CiberMouse*, des similarités et des différences apparaissent avec les spécifications de *maRTian Task*. Parmi les similarités, on observe que dans les deux concours :

- il est nécessaire de se déplacer d'un point A à un point B dans un environnement non connu a priori,

– il est nécessaire d'avoir une politique d'évitement des obstacles.

On observe cependant des différences entre les missions :

– un robot *CiberMouse* doit retourner à sa position initiale une fois la cible atteinte,
– un robot *CiberMouse* ne connait ni sa position initiale ni la position de la cible.
– un robot *CiberMouse* doit informer le simulateur de l'état dans lequel il se trouve.

Malgré ces différences, nous décidons de réutiliser le SAIM de *maRTian Task* contenant, dans l'application de contrôle, l'algorithme de recherche de chemin et l'algorithme d'évitement d'obstacles. Pourtant, *CiberMouse* demande également d'autres fonctionnalités telles que la recherche de la cible ou encore la communication de son état au simulateur.

Au niveau de la plateforme réelle (ici le simulateur), le corps virtuel du robot est de forme cylindrique et équipé de capteurs et d'actionneurs (cf. figure 4.19). Plus précisément, il est équipé de quatre capteurs d'obstacles, d'un capteur de lumière infrarouge donnant l'angle relatif de détection de la lumière, d'une boussole et d'un capteur de collision. Le robot comprend également deux moteurs diamétralement opposés permettant le contrôle de deux roues indépendantes de manière différentielle. Pour communiquer avec le simulateur et les utilisateurs, le robot comprend trois diodes électroluminescentes permettant de connaître son état. Les spécifications complètes du concours sont données dans [35]. Il faut remarquer que contrairement au concours précédent, le simulateur ne fournit ni la position du robot, ni son orientation. De plus, le nombre et la disposition des capteurs d'obstacles (bumper + sonars) sont différents. Enfin, la gestion des déplacements est également différente.

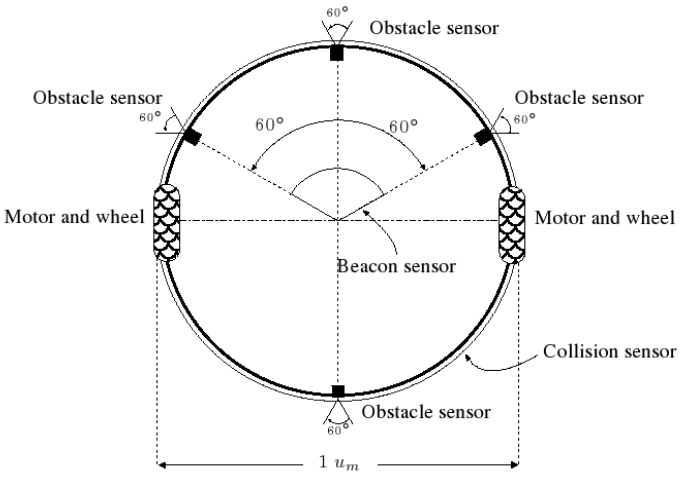


FIG. 4.19 – Infrastructure physique virtuelle du robot utilisé pour le concours `CiberMouse`

La section suivante présente et discute les modifications mineures apportées au SAIM. Ensuite, puisque le SAM est imposé par le simulateur, les changements apparaissent dans le connecteur complexe dont la réalisation est détaillée. Enfin, une troisième partie donne les bénéfices et limites mis en avant dans cet exercice de réutilisation.

4.6.2 Les modifications dans le SAIM

Le SAIM de *maRTian Task* a été modifié ; plus précisément, il a été étendu. En effet, en regardant les *sorties* identifiés dans le SAIM de *maRTian Task*, on se rend compte qu'aucune d'entre elles ne permet de communiquer les informations sur l'état du robot à l'environnement. De ce fait nous ajoutons une *sortie* : *Mission_ info* qui permet de renseigner sur l'état de la mission du robot (recherche de la cible, sur la cible, sur le chemin du retour, arrivé à destination). La deuxième extension qui fut nécessaire concerne la manière de manipuler cette nouvelle *sortie*. Pour ce faire, nous avons ajouté les états présentés précédemment à l'entrée existante `robot_state`. Malgré ces extensions, les fonctionnalités principales du SAIM telles que la recherche de chemin ou l'évitement d'obstacles n'ont pas été modifiées. Les autres fonctionnalités nécessaires pour réaliser les spécifications de *CiberMouse* ont donc été réalisées dans le connecteur complexe.

4.6.3 Le connecteur complexe

Le connecteur complexe, en charge de la liaison du SAIM et du SAM, a dans ce cas précis deux fonctions principales. Premièrement, il doit utiliser les services du SAM pour acquérir les *entrées*, et réaliser les *sorties* du SAIM. Deuxièmement, il doit "piloter" le SAIM afin de réaliser les fonctionnalités non directement liées à la recherche de chemin ou à l'évitement d'obstacles. Chacune de ces deux fonctions sont présentées au travers de :

1. la manière d'acquérir la position et l'orientation du robot,
2. la manière de piloter le SAIM par l'acquisition de l'*entrée* `goal_position`.

4.6.3.1 L'acquisition de la position du robot

La position absolue et l'orientation du robot (*entrée* notée `actual_position`) sont nécessaires à l'application. Malheureusement, le simulateur fourni pour le concours *CiberMouse* n'offre aucun pilote d'acquisition pouvant produire ces *entrées*. De ce fait, afin d'évaluer la position du robot, on crée un calcul en boucle ouverte (cf. figure 4.20). L'idée de la boucle ouverte est d'évaluer

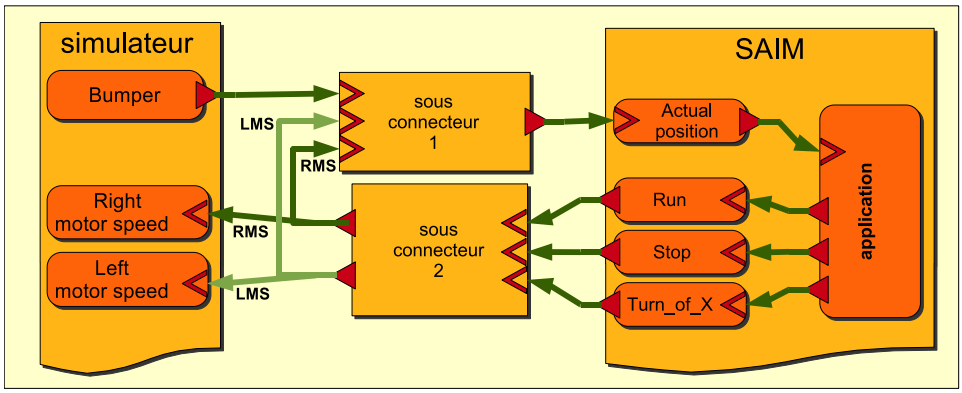


FIG. 4.20 – Acquisition de l'*entrée* `actual_position` en boucle ouverte

la position et l'orientation du robot d'après les *Commandes* en déplacement que l'on envoie au SAM. Pour cela, les trois *sorties* nommées `Run`, `Stop` et `Turn_of_X` sont, dans un premier temps, traduites par le sous connecteur 2 en *Commandes* de bas niveau pour les pilotes de réalisation `Right_motor_speed` et `Left_motor_speed`. Ces *Commandes*, notées RMS et LMS (*Right/Left Motor Speed*) sont également envoyées au sous connecteur 1, en charge de l'acquisition de l'*entrée* `actual_position`. Ainsi, dans le sous connecteur 1, les deux *Commandes* de bas niveau RMS et LMS sont agrégées par un composant de type `Interpret`. Ce composant, à l'aide de la position précédente, de la dynamique des moteurs et de la cinétique du robot (ici fournies par le simulateur), est capable de calculer la position résultante du robot. Afin de posséder un maximum de précision et éviter les dérives, les occurrences de la position sont calculées à la même cadence que le simulateur, c'est-à-dire à chacun de ces cycles. Cependant, afin d'éviter les coûts de communication de la position à chaque cycle, celle ci est filtrée par un composant de type `QoS_adapt` et n'est mise à jour dans `actual_position` qu'un cycle sur trois. Si le coût induit par un calcul de l'occurrence de position à chaque cycle est trop élevé, l'automate du composant de type `Interpret` devra être modifié afin de ne pas calculer l'occurrence à chaque cycle. Il devra tout de même connaître le temps écoulé entre deux calculs ainsi que les occurrences de RMS et LMS correspondantes afin de calculer l'évolution correspondante de `Actual_position`.

Pour ce calcul, nous devons également considérer les contraintes physiques parmi lesquelles le fait que le robot ne peut plus avancer lorsqu'il entre en collision avec un obstacle. Le flot d'informations en provenance du pilote d'acquisition des collisions (nommé `Bumper` sur la figure 4.20) renseigne sur les collisions ; c'est donc un flot d'informations devant être consommé par le sous connecteur 1.

Cette section a permis de détailler une partie du sous connecteur complexe permettant d'adapter le SAM proposé dans le cadre du concours *CiberMouse* aux *entrées* et aux *sorties* définies lors du concours *maRTian Task*. La section suivante détaille le pilotage du SAIM par le connecteur complexe afin de l'adapter aux besoins du concours *CiberMouse*.

4.6.3.2 Le pilotage du SAIM

Contrairement au concours *maRTian Task*, la position à atteindre dans l'environnement n'est pas connue a priori. La surface où se déplace le robot contient une source de lumière infrarouge en provenance de la position à atteindre mais ne pouvant pas être détectée aux travers des murs plus haut qu'elle. La première étape pour le robot consiste donc à rechercher la position de cette source lumineuse à l'aide du pilote d'acquisition du détecteur infrarouge. Ensuite, comme dans maRTian Task, le robot doit atteindre cette position. Une fois atteinte, le robot doit retourner à sa position initiale. On identifie donc trois phases successives :

1. Rechercher la cible.
2. Atteindre la cible.
3. Revenir à la position initiale

Chacune de ces trois phases peut être pilotée en modifiant le flot de données consommé par l'*entrée* `goal_position` du SAIM. C'est donc un seul connecteur, celui en charge de l'acquisition de l'*entrée* `goal_position`, qui va permettre de piloter le SAIM de *maRTian Task*.

Lors de la première phase, le sous connecteur en charge de l'acquisition de `goal_position` génère un flot de positions de manière à explorer la carte et ainsi espérer voir l'émission infrarouge de la cible au moins une fois. Pour ce faire, le sous connecteur doit consommer les flots d'informations suivant :

- le flot de données en provenance du pilote d'acquisition de détection de la cible,
- le flot de donnée en provenance du sous connecteur en charge de l'acquisition de la position et de l'orientation du robot.

Le sous connecteur doit également posséder une politique d'exploration de l'environnement lui permettant de générer une position à atteindre d'après les informations de ces flots.

Le passage à la phase deux s'effectue lorsque la cible a été détectée au minimum une fois. Le sous connecteur doit donc générer des positions qui vont permettre d'atteindre la cible. Lors de la détection de la cible, la donnée fournie est l'orientation relative de la cible, sans renseignement sur sa distance. Le sous connecteur doit conserver l'historique des positions dans lesquelles la cible a été vue et l'orientation associée afin de réaliser une triangularisation et ainsi fournir une position à atteindre de plus en plus précise. En plus des informations précédentes, ce sous connecteur doit consommer le flot de données représentant l'état du robot afin de savoir si la cible est atteinte ou non. Lorsque celle-ci est atteinte, la troisième phase commence. La position à atteindre est donc mise à jour par la position initiale du robot, préalablement sauvegardée par le sous connecteur.

4.6.3.3 Conclusion de *CiberMouse*

Ce travail a permis de montrer comment réutiliser un modèle SAIM sur un SAM différent en modifiant le connecteur complexe. En plus des différences au niveau de la plateforme réelle (ici le simulateur), des différences au niveau des spécifications de la mission à réaliser ont été gérées dans le connecteur complexe. Ce changement de plateforme et de spécification ont nécessité une modification du SAIM en ajoutant une *sortie* permettant de communiquer l'état de la mission du robot. Le modèle SAIM de *maRTian Task* ne possède effectivement pas de sortie permettant de communiquer son état. L'ajout de cette sortie est donc une amélioration du SAIM de *maRTian Task*. De plus, afin d'améliorer les futurs réutilisations, la *sortie* ne doit pas seulement communiquer les états de la mission important dans le contexte de *CiberMouse* mais tous les états de la mission et du robot. Ainsi, on remarque que la réutilisation d'un SAIM permet également d'améliorer celui-ci en le raffinant en incorporant l'expérience acquise au fur des réutilisations. L'objectif final serait de faire de ces composants des composants "métiers".

Il a été possible de piloter le SAIM de *maRTian Task* afin de l'adapter aux spécifications de *CiberMouse* en utilisant le sous connecteur en charge de l'acquisition de la position à atteindre. Cependant, on remarque que la complexité de l'ALM a, de ce fait, augmenté. En particulier, le sous connecteur décrit précédemment contient un cycle de vie pour gérer trois phases de fonctionnement différentes, un algorithme d'exploration de carte et un algorithme de triangularisation. On voit donc que la réutilisation d'un SAIM pour une utilisation différente de celle prévu a un coût qu'il est nécessaire d'évaluer pour tirer bénéfice de la réutilisation.

Le sous connecteur en charge de l'acquisition de la position à atteindre possède trois phases durant lesquelles sa politique de génération du flot de données est différente. On a donc un QoS différente pour chacun de ces flots. En généralisant ceci, on se rend compte que la QoS des *entrées*

et des *sorties* peut, dans certains cas être liée au cycle de vie du système. Il pourrait donc être intéressant de décrire plusieurs QoS pour une même *entrée* ou *sortie*, chacune correspondant à un cycle de vie différent du système.

Dans un autre registre, l'intérêt documentaire des modèles a été mis en avant lors de la réutilisation, une année après leur réalisation, des spécifications et de l'implémentation de *maRTian Task*. La vue gros grains des modèles, décrivant chacun un aspect différent du système (SAM, ALM, SAIM), permet une compréhension facilité du rôle de chacun des composants. La modification dans le code source associé est donc plus facile à localiser et à effectuer.

Enfin, la modélisation réalisée pour le contrôle du robot dans le cadre du concours *CiberMouse* nous a valu une place honorable de quatrième lors de la compétition.

4.7 Conclusion sur l'évaluation

Ce chapitre a présenté le cadre d'utilisation de SAIA au travers de plusieurs points. Le premier d'entre eux est celui de l'outil. La réalisation d'un outil a dans un premier temps permis de réaliser des modèles qui ne peuvent qu'être conformes au style architectural SAIA. Il a également permis de s'assurer de la cohérence entre les modèles SAM, ALM et SAIM pour la réalisation d'un modèle complet. Le point le plus important de la réalisation de l'outil est qu'il permet de lier cette approche aux approches de transformations de modèles et offre ainsi un pont vers d'autres approches et formalismes.

Le deuxième et le troisième points abordés présentent des méthodes qui permettent d'exploiter le style architectural. La première approche présente une suite de processus de modélisation permettant de construire chacun des modèles du style architectural proposé. Cette méthode montre comment développer puis déployer de manière sûre une application de contrôle. Pour ce faire, l'hypothèse est que la qualité de contrôle est fonction de l'application et de la QoS sur les *entrées* et les *sorties* Cette méthode est illustrée par un exemple qui permet de se figurer l'utilisation de SAIA pour réaliser la configuration logique d'un système. De plus l'exemple permet de montrer que lors de la dérivation de la qualité de contrôle en QoS sur la plateforme abstraite, il est intéressant d'utiliser une spécification précise de la QoS pour ne pas restreindre les déploiements ou pour ne pas surcharger le système en spécifiant une QoS trop restrictive.

La deuxième méthode est d'avantage focalisée sur la mise au point des paramètres temporels d'un système. Elle montre que SAIA, et en particulier la simulation exhaustive ainsi que la réalisation de contrat de QoS, permet de définir un ensemble de valeurs des paramètres pour lesquelles un contrat peut être établi.

Pour finir ce chapitre, deux mises en œuvre plus conséquentes ont été présentées. La première concerne le développement de l'application de contrôle de *maRTian Task*. Cette première mise en œuvre a permis de s'assurer de la faisabilité des modèles à l'exécution. SAIA permet la description d'une architecture logique, ainsi, lors de l'implémentation des modèles, la mise en place d'un modèle de tâches a donc été nécessaire. Elle nous a également permis de nous rendre compte de l'impact positif de la structure "gros grains" lors de l'implémentation puisque chacun

des composants peut être développé indépendamment des autres.

Enfin, la deuxième mise en œuvre a permis de présenter comment réutiliser un même SAIM sur deux SAM différents. Pour cela le SAIM de *maRTian Task* a été utilisé pour réaliser *CiberMouse*. Ainsi, nous avons vérifié qu'il est possible de connecter la plateforme abstraite à plusieurs plateformes réelles. Nous avons également pu apprécier l'intérêt de la séparation des préoccupations de part la facilité de reprise en main de la mise en œuvre réalisée l'année précédente.

L'ensemble des points abordés montre l'intérêt de baser une application de contrôle sur une plateforme abstraite plutôt que sur une plateforme réelle. Une plateforme abstraite définit un ensemble de plateformes réelles. Cet ensemble est restreint par une définition de la QoS permettant à l'application de contrôle d'atteindre ses objectifs de qualité de contrôle. La réalisation d'un contrat de QoS est alors nécessaire pour assurer un déploiement correct. La séparation des préoccupations ainsi réalisée et la structuration du système en composants permet de également de maîtriser la complexité du développement. De plus, les modèles réalisés agissent comme une documentation du système, facilitant ainsi les futures réutilisations.

5

Conclusion et Perspectives

Cette thèse propose un style architectural pour le développement d'une architecture logicielle dédiée aux systèmes de contrôle de processus. Les principaux objectifs de l'approche proposée sont de permettre le développement d'une application de contrôle indépendamment d'une plateforme de communication et de s'assurer ensuite de la correction du déploiement de cette application de contrôle sur une plateforme de communication spécifique. Par plateforme de communication on considère les capteurs, les actionneurs et les pilotes associés.

L'étude des architectures logicielles a permis d'identifier certains principes clés devant être mis en œuvre. Ces études ont mis en avant la nécessité de définir des styles architecturaux permettant d'une part d'appliquer certains principes de génie logiciel et d'autre part de restreindre les configurations réalisées afin qu'elles restent analysables. On remarque également aux vues de ces études que la description de la QoS requise et fournie par un composant permet d'utiliser celui-ci en respect de ses exigences extra-fonctionnelles. Ainsi, lors de la liaison entre deux composants, un contrat de QoS est établi si les deux composants peuvent être connectés tout en respectant les contraintes de QoS. Cependant, les contrats de QoS traités dans la littérature se focalisent sur les propriétés directement composables, décrites à l'aide de valeurs discrètes, contraignant ainsi l'expressivité. De plus, lors de la connexion entre deux interfaces, on utilise un connecteur possédant une "glue" pour pallier l'hétérogénéité entre les interfaces. Du constat de cette étude nous notons un point important, il est nécessaire de quantifier l'impact du connecteur sur la QoS lors de l'établissement du contrat, ce point n'est pas traité dès lors que la "glue" n'est pas un choix réalisé parmi une liste finie de comportements.

Puisque l'objectif est de permettre le développement d'une application de contrôle indépendamment d'une plateforme de communication spécifique, une étude à été réalisée sur les différents styles architecturaux permettant de s'abstraire d'une plateforme, au sens large du terme. Ceux-ci mettent clairement en avant l'utilisation de différentes couches logicielles reflétant différents niveaux d'abstraction vis-à-vis de la plateforme. On note aussi qu'il est possible de décrire une

plateforme abstraite afin de spécifier les besoins d'une application et ainsi définir un ensemble de plateformes potentiellement utilisables. Ces structurations sont rarement utilisées dans le domaine des systèmes visés car la correction de l'application est fortement liée aux performances de la plateforme.

Le style architectural proposé dans cette thèse propose d'utiliser une structuration en trois couches distinguant :

1. une application de contrôle et sa plateforme abstraite ;
2. une plateforme réelle ;
3. le connecteur complexe permettant de lier la plateforme abstraite et la plateforme réelle.

Puisque la correction de l'application est dépendante des performances de la plateforme, il faut restreindre l'ensemble des plateformes réelles utilisables par une description de la QoS sur la plateforme abstraite. On définit ainsi un ensemble de plateformes réelles qui, sous réserve de satisfaire la QoS et les services décrits dans la plateforme abstraite, peuvent être utilisées par l'application de contrôle concernée. On obtient ainsi une application non dépendante d'une plateforme réelle spécifique, cependant on est tout de même dépendant d'un ensemble de plateformes réelles, pas forcément connu a priori. Plus cet ensemble est grand, plus le niveau d'indépendance est élevé.

Afin d'obtenir un niveau d'indépendance élevé, il est important que la plateforme abstraite décrive les besoins de l'application de manière indépendante d'une technologie existante. Il est également important que la QoS ne restreigne pas drastiquement cet ensemble. Elle doit donc être exprimée d'une manière riche. Plutôt que de se baser sur des valeurs constantes ou des intervalles, nous décrivons un langage de la QoS décrit à l'aide d' ω-expression régulière. Enfin, dans le but de vérifier qu'une plateforme réelle appartient à l'ensemble des plateformes utilisables, des contrats de QoS sont établis entre chaque composant de la plateforme abstraite et de la plateforme réelle. Les contrats sont établis si la QoS fournie satisfait la QoS fournie, une relation de satisfaction doit donc être définie pour savoir si l'établissement d'un contrat peut avoir lieu ou non.

Afin de pouvoir réaliser les différentes étapes décrites précédemment, le style architectural SAIA décrit des composants types pour spécifier chacune des trois couches identifiées et des règles de structuration permettant de s'assurer d'obtenir un connecteur complexe analysable, permettant ainsi l'établissement de contrats de QoS.

L'évaluation de SAIA a été réalisée au travers de divers aspects : l'outillage, les méthodes et plusieurs mises en œuvre. Elle a permis de caractériser les apports et les limites du style architectural proposé. Au travers de ces diverses mises en œuvre il a été possible d'extraire plusieurs bonnes propriétés de SAIA.

Au niveau fonctionnel, il a été possible de réutiliser le même SAIM sur plusieurs plateformes réelles grâce à la spécification de la plateforme abstraite. De plus, lors d'une modification dans la plateforme réelle ou dans la plateforme réelle, les changements interviennent localement au sein du ou des sous connecteur(s) concerné(s). Le fait de concentrer les modifications à réaliser dans un composant est une aide précieuse pour le développeur. La séparation du connecteur complexe en sous connecteurs permet également au développeur de se concentrer sur un composant spécifique du système. Puisque la plateforme abstraite agit comme une interface entre la plateforme réelle et l'application, il est également possible de réaliser des changements dans l'application de contrôle

sans pour autant que ceux-ci soient propagés dans tout le système. Il est également possible de développer l'application de contrôle et la connexion à une plateforme réelle en parallèle.

Au niveau extra-fonctionnel, l'utilisation d'un langage de QoS permet d'être plus expressif, cependant, il est difficile de savoir intuitivement si un contrat peut être établi ou non. SAIA offre donc une chaîne d'outil permettant d'automatiser l'établissement de contrat de QoS. De plus, les contrats sont établis sous connecteur par sous connecteur. Ainsi, il est possible de se focaliser sur le paramétrage d'un sous connecteur jusque l'établissement d'un contrat de QoS. Ce paramétrage est facilité par la possibilité de réaliser un ensemble d'analyses de manière automatique.

Perspectives

Les résultats obtenus lors de ce travail de thèse permettent d'entrevoir de nombreuses perspectives de recherche que nous exposons ici selon trois grands axes :

Extension des modèles

- SAIA est basé sur un modèle à composants ad-hoc afin de s'affranchir des contraintes implicites inhérentes aux modèles à composants existant. De plus, ceci nous permettait de choisir une sémantique opérationnelle formelle. Il serait intéressant d'étudier dans quelle mesure SAIA peut être utilisée sur un modèle à composant existant. La première étape serait alors de rendre explicite le style architectural sous-jacent au modèle à composant sélectionné. La seconde étape serait d'évaluer la propension du style architectural ainsi rendu explicite à accueillir la sémantique opérationnelle formelle de IF. Enfin, il serait nécessaire d'augmenter le modèle à composant par les types de composants et les contraintes structurelles définis dans SAIA.
- Les différentes analyses, basées sur IF dans SAIA, ont été effectuées en se basant sur un temps discret. Il est donc nécessaire de réaliser une description de la QoS comme un ensemble de valeurs discrètes. Afin d'étendre le pouvoir d'expression de la QoS, il serait intéressant d'évaluer l'impact d'une description du langage de la QoS comprenant des intervalles continus. Il faudrait alors réaliser les analyses en temps continu (IF en temps continu ou UPPAAL) et vérifier si les relations de satisfaction proposées sont toujours valides, en particulier l'équivalence de trace.
- Les caractéristiques de QoS prises en compte dans SAIA ont été choisies pour leur impact sur la qualité de contrôle et la stabilité des systèmes de contrôle de processus. Cependant, d'autres caractéristiques ayant un impact sur la qualité de contrôle peuvent être envisagées. On peut citer deux types de caractéristiques différentes : les caractéristiques temporelles telles que la gigue et les caractéristiques non temporelles telles que la précision des données. La première étape de cette extension consisterait donc à choisir une technique de spécification pour chaque caractéristique de QoS. Ensuite, une relation de satisfaction entre la QoS fournie et la QoS requise doit être formalisée. Enfin, il faudrait évaluer l'impact de l'incorporation de chacune des caractéristiques sur l'analyse. En particulier, les caractéristiques non temporelles telles que la précision des données pourraient demander la création d'un modèle reflétant l'évolution de l'environnement.

- Dans SAIA la spécification de la QoS dans la plateforme abstraite est réalisée pour chaque *entrée* et pour chaque *sortie*. Cependant, cette description est statique quelque soit le mode de fonctionnement (normal, dégradé, etc.) de l'application de contrôle. Certaines *entrées* et/ou *sorties* devraient pouvoir spécifier une qualité de service différente en fonction du cycle de vie de l'application de contrôle. Il serait alors nécessaire de décrire le cycle de vie de l'application de contrôle ainsi que les liens qui lient ce cycle de vie à la QoS de la plateforme abstraite. Enfin, il faudrait investiguer l'impact du cycle de vie sur le connecteur complexe et sur l'établissement des contrats de QoS.

Mise en œuvre

- SAIA permet de décrire une architecture logique. On peut ainsi réaliser des analyses avant de traiter les détails de l'implémentation. La plateforme d'exécution et l'effet de l'ordonnancement sont abstraits par les budgets temporels. Afin de réaliser l'implémentation exécutable d'un modèle SAIA, il est nécessaire de préciser le langage de programmation, l'utilisation ou non d'un intergiciel ou encore le modèle de tâches. Il serait alors intéressant d'investiguer le raffinement de SAIA afin de produire une architecture opérationnelle en partant de l'architecture logique. Dans la même idée que Accord [75], une solution serait de caractériser une plateforme d'exécution multitâches réelle (du type d'un système d'exploitation temps réel) et une machine virtuelle d'exécution multitâche. Alors, dans la même idée que SAIA, la machine virtuelle serait vue comme une plateforme d'exécution abstraite. Ensuite le travail consisterai à : formaliser les caractéristiques de QoS pertinentes, formaliser la description de la QoS requise et fournie et formaliser la relation de satisfaction pour chaque caractéristique de QoS. Ainsi, on devrait pouvoir caractériser la connexion entre la plateforme réelle et la plateforme abstraite et, si nécessaire, expliciter les contraintes et méthodes permettant d'envisager l'établissement d'un contrat de QoS.

Extensions du domaine

- Aucun des types de composants définis dans SAIA ne permet de modéliser les effets de réseaux de terrains au sein d'un système de contrôle. Cependant, de plus en plus de systèmes de contrôle de processus, à l'image des systèmes automobiles, utilisent un réseau de communication. L'utilisation de ces réseaux peut s'effectuer entre les capteurs, les actionneurs et l'application de contrôle ou entre différentes applications de contrôle. Afin de permettre la modélisation de tels systèmes, il serait intéressant, dans la même idée que [77], d'étudier l'impact des différents réseaux de terrains (CAN, TTA, ...) sur les caractéristiques de QoS décrites dans SAIA. Ceci devrait permettre de caractériser de manière abstraite l'effet des communications dans le but de réaliser les analyses décrites dans SAIA, dans le cadre des systèmes distribués.
- Dans SAIA, les interprétations nécessaires entre les informations de la plateforme abstraite et celles de la plateforme réelle sont réalisées statiquement pendant la phase de modélisation. Dans les systèmes où les contraintes temporelles sont moindres et les systèmes ouverts (PDA, téléphones, ...) l'arrivée et le déploiement d'une application devrait se faire de manière transparente sur plusieurs plateformes. L'utilisation d'une plateforme abstraite semble être une bonne solution pour la spécification des besoins sans pour autant spécifier une technologie particulière. Dans le but de connecter la plateforme abstraite à la plateforme réelle, il serait alors nécessaire de définir une base ontologique

regroupant la sémantique de la plateforme abstraite et de la plateforme réelle ainsi que des interprétations préétablies permettant de connecter les deux plateformes. Il serait alors nécessaire d'utiliser un intergiciel spécifique permettant de gérer, à l'exécution, la description ontologique des plateformes ainsi que le choix de l'interprétation pour l'établissement de la connexion entre les plateformes.

Communications effectuées dans le cadre de cette thèse

Conférences

- Julien DeAntoni and Jean-Philippe Babau. *A MDA approach for systems dedicated to process control.* Eleventh IEEE International Embedded and Real Time Computing Systems and Applications (RTCSA'05), pages 567-570, Hong-Kong 2005.
- Julien DeAntoni and Jean-Philippe Babau. *A MDA-based approach for real time embedded systems simulation.* Nineth IEEE International Symposium on Distributed Simulation and Real-Time Applications (DS-RT'05), pages 257-264. Montréal 2005.
- Julien DeAntoni and Jean-Philippe Babau. *SAIA : Sensors/Actuators Independent Architecture - a showcase through martian task specification.* Proceedings of the ERTSI 2005 - Embedded Real Time Systems Implementation Workshop, held in conjunction with 26th IEEE International Real-Time Systems Symposium, pages 43-50, Miami 2005. http :// www.cs.york.ac.uk/ftpdir/reports/YCS-2005-397.pdf.
- Julien DeAntoni and Jean-Philippe Babau. *Model driven engineering method for SAIA architecture design.* 3th Ingénierie Dirigée par les modèles (IDM'06), pages 109-122, Lille 2006.
- Julien DeAntoni and Jean-Philippe Babau. *Cibermouse design : A case study for SAIA model reuse.* Proceedings of the Cibermouse Team reports, a satellite event of the 27th IEEE International Real-Time Systems Symposium, pages 9-12, Rio de Janeiro 2006.
- Julien DeAntoni, Jean-Philippe Babau. *SAIA : safe deployment of sensors based real time application.* Workshop on Models and Analysis for Automotive Systems (held in conjunction with the 27th IEEE International Real-Time Systems Symposium) ; pages 45-48. Rio de Janeiro 2006
- Julien DeAntoni, Fabrice Jumel and Jean-Philippe Babau. *tech_ report20070130 : A simple simulink robot simulator.* Technical report, INSA-Lyon, 2007.
- Jean-Philippe Babau and Julien DeAntoni. *Architectures logicielles pour les systèmes embarqués temps réel.* 5th École d'été temps réel, pages 75-94, Nantes 2007.

Séminaire

- SAIA : un style architectural pour assurer l'indépendance vis-à-vis d'entrées / sorties soumises à des contraintes temporelles. Séminaire dans le cadre des journées générales du projet EmSoC-Recherche, 18 octobre 2007, Aussois.
- SAIA : Sensor / Actuator Independent Architecture. Séminaire dans le cadre des journées composants du projet EmSoC recherche, 1 juin 2007, Grenoble.
- SAIA : Sensor / Actuator Independent Architecture. Séminaire dans le cadre du projet REVE : safe Reuse of Embedded components in heterogeneous enVironmEnts. 25 Avril 2007, Lyon
- SAIA : a Sensor / Actuator Independent Architecture. Présentation poster à MDE 2006 (Model Driven Embedded), septembre 2006, Aber Wrac'h
- SAIA : a Sensor / Actuator Independent Architecture. Présentation poster aux journées EmSoC recherche, 8-9 juin 2006, Villars de Lens
- Une approche MDA pour les systèmes temps réel. Séminaire thésard au laboratoire CITI. avril 2005, Lyon
- Démonstration de la modélisation et implémentation réalisé dans le cadre du concours Josefil. septembre 2004, Aber Wrac'h

Bibliographie

[1] M. Agrawal, S. Cooper, L. Graba, and V. Thomas. An open software architecture for high-integrity and high-availability avionics. *Digital Avionics Systems Conference, 2004. DASC 04. The 23rd*, 2, 2004.

[2] Robert Allen. *A Formal Approach to Software Architecture*. PhD thesis, Carnegie Mellon, School of Computer Science, January 1997. Issued as CMU Technical Report CMU-CS-97-144.

[3] JP Almeida, R. Dijkman, M. van Sinderen, and LF Pires. On the notion of abstract platform in MDA development. *Enterprise Distributed Object Computing Conference, 2004. EDOC 2004. Proceedings. Eighth IEEE International*, pages 253–263, 2004.

[4] R. Alur and DL Dill. Automata for modeling real-time systems. *Proceedings of the seventeenth international colloquium on Automata, languages and programming table of contents*, pages 322–335, 1990.

[5] ARTIST. *Adaptive Real-Time Systems for Quality of Service Management*, pages part 1 – chapter 3. 2003.

[6] ARTIST. Adaptive real-time systems for quality of service management. *Draft IST-2001-34820*, 2003.

[7] AUTOSAR. Automotive open system architecture. http ://www.autosar.org, 2007.

[8] Jean-Philippe Babau and Sebastien Gerard. *Model-Driven Schedulability Analysis*, chapter 9. Hermes Science publishing, 2005.

[9] Len Bass, Paul Clements, and Rick Kazman. *Software Architecture in Practice, Second Edition*. Addison-Wesley Professional, April 2003.

[10] Ananda Basu, Marius Bozga, and Joseph Sifakis. Modeling heterogeneous real-time components in bip. In *SEFM '06 : Proceedings of the Fourth IEEE International Conference on Software Engineering and Formal Methods*, pages 3–12, Washington, DC, USA, 2006. IEEE Computer Society.

[11] Iain Bate, Peter Nightingale, and Anton Cervin. Establishing timing requirements and control attributes for control loops in real-time systems. *15th Euromicro Conference on Real-Time Systems (ECRTS'03)*, 00 :121, 2003.

[12] Belgacem Ben-Hedia, Fabrice Jumel, and Jean-Philippe Babau. Formal evaluation of quality of service for data acquisition systems. *Forum on specification and Design Language*, 2005.

[13] G. Berry and R. Sethi. From regular expressions to deterministic automata. *Theoretical Computer Science*, 48(1) :117–126, 1986.

[14] B. Berthomieu, P-O. Ribet, and F. Vernadat. Towards the verification of real-time systems in avionics : The cotre approach. *8th international workshop on formal method for industrial critical systems (FMICS)*, 2003.

[15] Antoine Beugnard, Jean-Marc Jézéquel, Noël Plouzeau, and Damien Watkins. Making component contract aware. *IEEE computer, 32(7)*, pages 38–45, 1999.

[16] T.A. Bishop and R.K. Karne. A SURVEY of MIDDLEWARE.

[17] Marius Bozga, Suzanne Graf, and L. Mounier. If-2.0 : A validation environment for component-based real time systems. *In ed. Brinksma, K.G. Larsen (Eds) Proceedings of CAV'02*, 2002.

[18] F. Budinsky. *Eclipse Modeling Framework*. Addison-Wesley Professional, 2003.

[19] J.N. Buxton and B. Randell. Software engineering techniques : Report on a conference sponsored by the nato science committee rome italy the 27-31th oct. 1969. Scientific Affairs Division, NATO, Brussels, 1970.

[20] Eric Cariou, Antoine Beugnard, and Jean-Marc Jézéquel. An architecture and a process for implementing distributed collaborations. In *EDOC*, pages 132–143. IEEE Computer Society, 2002.

[21] Tarak Chaari and Frédérique Laforest. Sefagi : Simple environment for adaptable graphical interfaces - generating user interfaces for different kinds of terminals. In Chin-Sheng Chen, Joaquim Filipe, Isabel Seruca, and José Cordeiro, editors, *ICEIS (1)*, pages 232–237, 2005.

[22] Kenneth Chan. Formal proofs for qos-oriented transformations. *edocw*, 0 :41, 2006.

[23] D. D. Clark and D. L. Tennenhouse. Architectural considerations for a new generation of protocols. *SIGCOMM Comput. Commun. Rev.*, 20(4) :200–208, 1990.

[24] David D. Clark and David L. Tennenhouse. Architectural considerations for a new generation of protocols. In *SIGCOMM*, pages 200–208, 1990.

[25] J. Coutaz. Abstractions for user interface design. *Computer*, 18(9) :21–34, 1985.

[26] J. Coutaz. Abstractions for user interface toolkits. *Foundation for Human-Computer Communication, Elsevier Science, North-Holland*, 1986.

[27] J. Coutaz. *Interfaces homme-ordinateur : conception et réalisation*, pages 139–145. Dunod, 1990.

[28] J. Coutaz. *Interfaces homme-ordinateur : conception et réalisation*. Dunod, 1990.

[29] J. Coutaz. *Interfaces homme-ordinateur : conception et réalisation*, pages 161–186. Dunod, 1990.

[30] I. Crnkovic, M. Larsson, and O. Preiss. Concerning Predictability in Dependable Component-Based Systems : Classification of Quality Attributes. *lecture notes in computer science*, 3549 :257, 2005.

[31] Ivica Crnkovic. *Building Reliable Component-Based Software Systems*. Artech House, Inc., Norwood, MA, USA, 2002.

[32] Ivica Crnkovic. Building reliable component-based software systems. pages 189–190, 2002.

[33] K. Czarnecki and S. Helsen. Feature-based survey of model transformation approaches. *IBM Systems Journal*, 45(3) :621–645, 2006.

[34] Julien DeAntoni, Jumel Fabrice, and Jean-Philippe Babau. tech_report20070130 : A simple simulink robot simulator. Technical report, INSA-Lyon, 2007.

[35] Telecomunicações e Inform_'atica Universidade de Aveiro Departamento de Electrónica. Ciber-mouse : Rules and technical specifications. http ://www.ieeta.pt/ lau/web_ ciberRTSS/docs/ciberRTSSrules.pdf, 2006.

[36] Frank DeRemer and Hans Kron. Programming-in-the large versus programming-in-the-small. In *Proceedings of the international conference on Reliable software*, pages 114–121, New York, NY, USA, 1975. ACM Press.

[37] Marcio S. Dias and Marlon E. R. Vieira. Software architecture analysis based on statechart semantics. In *IWSSD '00 : Proceedings of the 10th International Workshop on Software Specification and Design*, page 133, Washington, DC, USA, 2000. IEEE Computer Society.

[38] Edsger W. Dijkstra. The structure of the THE-multiprogramming system. *Commun. ACM*, 11(5) :341–346, 1968.

[39] dSPACE. Powerful tools for controller development. http ://www.dspaceinc.com/ww/en/inc/home/products.cfm ?nv=n2, 2007.

[40] E. Durand. Description et vérification d'architecture temps réel : Clara et les réseaux de petri temporisés. *Thèse de doctorat, école Centrale de Nantes France*, 2004.

[41] esterel technology. Scade suite (tm). http ://www.esterel-technologies.com/products/scade-suite/, 2007.

[42] P. Farail and P. Gaufillet. The cotre project : How to model and verify real time architecture ? *2nd European Congress on Embedded Real Time Software (ERTS'2004)*, 2004.

[43] P. Farail, P. Gaufillet, A. Canals, C. Le Camus, D. Sciamma, P. Michel, X. Crégut, and M. Pantel. The TOPCASED project : a Toolkit in OPen source for Critical Aeronautic SystEms Design. *Embedded Real Time Software (ERTS'06), Toulouse, France*, pages 25–27, 2006.

[44] Jean-Philippe Fassino, Jean-Bernard Stefani, Julia L. Lawall, and Gilles Muller. Think : A software framework for component-based operating system kernels. In *USENIX Annual Technical Conference, General Track*, pages 73–86, 2002.

[45] Sébastien Faucou, Anne-Marie Déplanche, and Yvon Trinquet. An ADL centric approach for the formal design of real time systems. *In Architecture description language, IFIP*, pages 67–82, 2004.

[46] J.C. Fernandez, H. Garavel, A. Kerbrat, L. Mounier, R. Mateescu, and M. Sighireanu. CADP-A Protocol Validation and Verification Toolbox. *Proceedings of the 8th International Conference on Computer Aided Verification*, pages 437–440, 1996.

[47] Jean-Claude Fernandez. Aldébaran : a tool for verification of communicating processes. *technical report SPECTRE c14, LGIIMAG*, 1989.

[48] Roy Thomas Fielding. *Architectural styles and the design of network-based software architectures.* PhD thesis, 2000. Chair-Richard N. Taylor.

[49] William B. Frakes and Kyo Kang. Software reuse research : Status and future. *IEEE Trans. Softw. Eng.*, 31(7) :529–536, 2005.

[50] Robert B. France, Dae-Kyoo Kim, Sudipto Ghosh, and Eunjee Song. A uml-based pattern specification technique. *IEEE Transactions on Software Engineering*, 30(3) :193–206, 2004.

[51] J. Fredriksson, K. Sandstrom, and M. Akerholm. Optimizing resource usage in component-based real-time systems. *Proc. ACM International Symposium on Component-Based Software Engineering (CBSE)*, pages 49–65, 2005.

[52] David Garlan. Software architecture : a roadmap. In A. Finkelstein, editor, *The Future of Software Engineering*. ACM Press, 2000.

[53] David Garlan. Formal modeling and analysis of software architecture : Components, connectors, and events. In *SFM*, pages 1–24, 2003.

[54] David Garlan, Robert Allen, and John Ockerbloom. Exploiting style in architectural design environments. In *SIGSOFT '94 : Proceedings of the 2nd ACM SIGSOFT symposium on Foundations of software engineering*, pages 175–188, New York, NY, USA, 1994. ACM Press.

[55] David Garlan and D. Shaw. IEEE recommanded practice for architectural description of software-intensive systems. *ANSI/IEEE Standard 1471*, 2001.

[56] David Garlan and Mary Shaw. An introduction to software architecture. Technical report, Pittsburgh, PA, USA, 1994.

[57] T. Genssler. Pecos in a nutshell, 2002.

[58] Carlo Ghezzi, Mehdi Jazayeri, and Dino Mandrioli. *Fundamentals of software engineering*. Prentice-Hall, Inc., Upper Saddle River, NJ, USA, 1991.

[59] ETAS group. Ascet software products. http ://www.etas.com/en /products/ ascet_software_products.php, 2007.

[60] Z. Gu and Z. He. Real-time scheduling techniques for implementation synthesis from component-based software models. *Proc. ACM SIGSOFT International Symposium on Component-Based Software Engineering (CBSE)*, 2005.

[61] T.A. Henzinger, B. Horowitz, and C.M. Kirsch. Giotto : a time-triggered language for embedded programming. *Proceedings of the IEEE*, 91(1) :84–99, 2003.

[62] ISIS. The generic modeling environment (gme), 2005. ISIS : Institute for Software Integrated Systems. Vanderbilt University.

[63] D. Isovic and C. Norstrm. Components in real-time systems, 2002.

[64] Jean-Marc Jézéquel, Olivier Defour, and Noël Plouzeau. An mda approach to tame component based software development. In Frank S. de Boer, Marcello M. Bonsangue, Susanne Graf, and Willem P. de Roever, editors, *FMCO*, volume 3188 of *Lecture Notes in Computer Science*, pages 260–275. Springer, 2003.

[65] Jean-Marc Jézéquel, Sébastien Gérard, and Benoît BAUDRY. *L'ingénierie dirigée par les modèles. Au-delà du MDA – Chapitre 3. Le génie logiciel et l'IDM : une approche unificatrice par les modèles.* Hermès – Lavoisier.

[66] Fabrice Jumel. Definition and management of a quality of service for real time applications *(in french). thesis in LORIA laboratory, Nancy*, 2003.

[67] Rick Kazman, Gregory Abowd, Len Bass, and Paul Clements. Scenario-based analysis of software architecture. *IEEE Software*, 13(6) :47–55, November 1996.

[68] Rick Kazman, Leonard J. Bass, Mike Webb, and Gregory D. Abowd. SAAM : A method for analyzing the properties of software architectures. In *International Conference on Software Engineering*, pages 81–90, 1994.

[69] Dae-Kyoo Kim, Robert France, Sudipto Ghosh, and Eunjee Song. Using role-based modeling language (rbml) to characterize model families. In *ICECCS '02 : Proceedings of the Eighth International Conference on Engineering of Complex Computer Systems*, page 107, Washington, DC, USA, 2002. IEEE Computer Society.

[70] M. Klein and R. Kazman. Attribute based architectural styles. Technical report, 1999.

[71] M. Klein, R. Kazman, L. Bass, J. Carriere, M. Barbacci, and H. Lipson. Attributebased architecture styles. In *Software Architecture, Proceedings of the First Working IFIP Conference on Software Architecture (WICSA1)*, pages 225–243. Kluwer Academic Publishers, 1999.

[72] M.H. Klein. *A Practitioner's Handbook for Real-Time Analysis : Guide to Rate Monotonic Analysis for Real-time Series*. Kluwer Academic Publishers, 1993.

[73] H. Kopetz, A. Damm, C. Koza, M. Mulazzani, W. Schwabl, C. Senft, and R. Zainlinger. Distributed fault-tolerant real-time systems : the Mars approach. *Micro, IEEE*, 9(1) :25–40, 1989.

[74] Glenn E. Krasner and Stephen T. Pope. A cookbook for using the model-view controller user interface paradigm in smalltalk-80. *J. Object Oriented Program.*, 1(3) :26–49, 1988.

[75] Agnès Lanusse, Sébastien Gérard, and FranÃ§ois Terrier. Real-time modeling with uml : The accord approach. volume 1618, pages 319–335. Springer, 1999.

[76] Kim G. Larsen, Paul Pettersson, and Wang Yi. UPPAAL in a Nutshell. *Int. Journal on Software Tools for Technology Transfer*, 1(1–2) :134–152, October 1997.

[77] Feng-Li Lian. Analysis, design, modeling, and control of networked control systems. *Ph.D. Dissertation, The University of Michigan*, 2001.

[78] Gabor Madl, Sherif Abdelwahed, and Douglas Schmidt. Verifying distributed real time properties of embedded systems via graph transformation and model checking. *Real-Time Systems : The international journal of Time-critical Computing systems, Volume 33*, 2006.

[79] R. Marvie and M. Nebut. Processus de Modélisation Incrémentaux. *Ingénierie Dirigée par les Modèles*, pages 139–154, 2006.

[80] Mathworks Inc. MATLAB. *http ://www.mathworks.com/products/matlab/*, 2005.

[81] Mathworks Inc. real time workshop. *http ://www.mathworks.com/products/rtw/*, 2005.

[82] Mathworks Inc. SIMULINK. *http ://www.mathworks.com/products/simulink/*, 2005.

[83] Nenad Medvidovic, Peyman Oreizy, Jason E. Robbins, and Richard N. Taylor. Using object-oriented typing to support architectural design in the c2 style. In *SIGSOFT '96 : Proceedings of the 4th ACM SIGSOFT symposium on Foundations of software engineering*, pages 24–32, New York, NY, USA, 1996. ACM Press.

[84] Nenad Medvidovic and Richard N. Taylor. A classification and comparison framework for software architecture description languages. *Software Engineering*, 26(1) :70–93, 2000.

[85] Microsoft. Review of windows 3.1 architecture. *http ://www.microsoft.com/technet/archive/wfw/*$1_c h2.mspx$, 2005.

[86] Microsoft. .NET. http ://www.microsoft.com/net, 2006.

[87] Microsoft and Steven Cherry. Robot, incorporated. *IEEE spectrum*, august 2007.

[88] Sun microsystems. JSR 153 : Enterprise JavaBeans 2.1. Technical Report, Java Community Process, 2003.

[89] RT Monroe, A. Kompanek, R. Melton, and D. Garlan. Architectural styles, design patterns, and objects. *Software, IEEE*, 14(1) :43–52, 1997.

[90] A. Muth, T. Kolloch, T. Maier-Komor, and G. Farber. An evaluation of code generation strategies targeting hardware forthe rapid prototyping of SDL specifications. *Rapid System Prototyping, 2000. RSP 2000. Proceedings. 11th International Workshop on*, pages 134–139, 2000.

[91] P. Naur and B. Randell. Software engineering techniques : Report on a conference sponsored by the nato science committees garmisch germany 7-11 oct. 1968. Scientific Affairs Division, NATO, Brussels, 1969.

[92] D.I. Norm and G. Berlin. Profibus Standard : DIN 19245, 1995. *Two volumes.*

[93] FIP Normes. NF C46-601 to NF C46-607. *Union Technique de l'Electricité, AFNOR*, 1990.

[94] OMG. Object management group. *http ://www.omg.org*, 2003.

[95] OMG-MDA. Model driven architecture guide v1.0.1. *http ://www.omg.org/mda*, 2003.

[96] OMG-OCL. UML 2.0 OCL specification. *http ://www.omg.org/cgi-bin/doc ?ptc/2005-06-06*, 2005.

[97] OMG-UML. Final ftf report mof 2.0 core and uml 2.0 infrastructure finalization task force. *http ://www.omg.org*, 2004.

[98] The Open Service Gateway initiative. OSGI release platform 4. Technical Report, 2006.

[99] P.H.Feiler, B. Lewis, and Steve Vestal. the sae avionic architecture description language (aadl) standard : A basis for model-based architecture driven embedded systems engineering. *RTAS Workshop on Model-Driven Embedded Systems*, 2003.

[100] P. Pleinevaux, J.D. Decotignie, and L. EPFL. Time critical communication networks : field buses. *Network, IEEE*, 2(3) :55–63, 1988.

[101] project EAST-EEA. Definition of language for automotive embedded electronic architecture. *Version 1.02*, 2004.

[102] E. Robles and J. Held. A comparison of windows driver model latency performance on windows nt and windows 98. *Proc. OSDI third symposium*, 1999.

[103] Mendel Rosenblum. The reincarnation of virtual machines. *ACM queue :"Virtual Machine"; number 5; volume 2*, 2004.

[104] M. Sanfridson. Timing Problems in Distributed Real-Time Computer Control Systems. *Mechatronics Lab, Dept. of Machine Design, Royal Inst. of Technology, Stockholm*, 2000.

[105] M. Sanfridson, M. Törngren, and J. Wikander. The Effect of Randomly Time-Varying Sampling and Computational Delay. *In Proceedings of the IFAC world congress, Prague*, 2005.

[106] B. Selic. On the Semantic Foundations of Standard UML 2.0. *Lecture Notes in Computer Science*, 3185 :181–199, 2004.

[107] G.R. Sell. Smooth Linearization Near a Fixed Point. *American Journal of Mathematics*, 107(5) :1035–1091, 1985.

[108] Mary Shaw, Robert DeLine, Daniel V. Klein, Theodore L. Ross, David M. Young, and Gregory Zelesnik. Abstractions for software architecture and tools to support them. *Software Engineering*, 21(4) :314–335, 1995.

[109] Mary Shaw and David Garlan. *Software Architecture : Perspectives on an Emerging Discipline.* Prentice-Hall, 1996.

[110] Y. Shoham et al. Agent-Oriented Programming. *Artificial Intelligence*, 60(1) :51–92, 1993.

[111] ISO Standard. 11898 : Road Vehicles - Interchange of Digital Information - Controller Area Network (CAN) for High-Speed Communication. *International Standards Organization, Switzerland, November*, 1993.

[112] J.A. Stankovic. VEST-A Toolset for Constructing and Analyzing Component Based Embedded Systems. *Proceedings of the First International Workshop on Embedded Software*, pages 390–402, 2001.

[113] David B. Stewart, Richard A. Volpe, and Pradeep K. Khosla. Design of dynamically reconfigurable real-time software using port-based objects. *Software Engineering*, 23(12) :759–776, 1997.

[114] B. Stroustrup. Language-technical aspects of reuse. In *ICSR '96 : Proceedings of the 4th International Conference on Software Reuse*, page 11, Washington, DC, USA, 1996. IEEE Computer Society.

[115] sun. Java 2 platform standard edition development kit 5.0. http ://www.java.sun.com.

[116] Clemens Szyperski. *Component Software : Beyond Object-Oriented Programming*. Addison-Wesley Professional, December 1997.

[117] Q. Tan, A. Petrenko, and G. Bochmann. *Testing Trace Equivalence for Labeled Transition Systems*. Université de Montréal, Département d'informatique et de recherche opérationnelle, 1995.

[118] A.S. Tanenbaum. *Modern operating systems*. Prentice-Hall, Inc. Upper Saddle River, NJ, USA, 1992.

[119] A.S. Tanenbaum. *Modern operating systems – Input Output chapter*, pages 205–239. Prentice-Hall, Inc. Upper Saddle River, NJ, USA, 1992.

[120] FrameIP TcpIP. Les 7 couches du modèle osi. http ://www.frameip.com/osi/ #4.2_-_Ce_n_la_bonne_technologie, 2003.

[121] Bedir Tekinerdogan. Asaam : Aspectual software architecture analysis method. *wicsa*, 00 :5, 2004.

[122] J.P. Tolvanen and M. Rossi. MetaEdit+ : defining and using domain-specific modeling languages and code generators. *Conference on Object Oriented Programming Systems Languages and Applications*, pages 92–93, 2003.

[123] Jean-Charles Tournier. *Qinna : une architecture à base de composants pour la gestion de la qualité de service dans les systèmes embarqués mobiles*. PhD thesis, Institut National des Sciences Appliquées de Lyon, July 2005.

[124] Rob van Ommering, Frank van der Linden, Jeff Kramer, and Jeff Magee. The koala component model for consumer electronics software. *Computer*, 33(3) :78–85, 2000.

[125] Steve Vestal. Metah user's manual - version 1.27. Honeywell Technology center - Minneapolis, MN, USA, 1998.

[126] Anders Wall. A formal approach to analysis of software architectures for real-time systems. Technical report, September 2000.

[127] Kurt Wallnau, Judith Stafford, Scott Hissam, and Mark Klein. On the relationship of software architecture to software component technology. *Proc. 6th Workshop on Component-Oriented Programming*, 2001.

[128] Kurt C. Wallnau. Volume iii : A technology for predictable assembly from certifiable components (pacc). Technical Report CMU/SEI-2003-TR-009, Carnegie Mellon University, Pittsburgh, OH, USA, April 2003.

[129] B. Wittenmark, J. Nilsson, and M. Torngren. Timing problems in real-time control systems. *American Control Conference, 1995. Proceedings of the*, 3, 1995.

[130] Sergio Yovine. Kronos : A verification tool for real-time systems. (kronos user's manual release 2.2).

[131] H. ZIMMERMANN. OS1 Reference Model-The IS0 Model of Architecture for Open Systems Interconnection. *IEEE TRANSACTIONS. ON COMMUNICATIONS*, 28(4) :425, 1980.

Table des figures

Liste des tableaux

Résumé

Du fait de leur complexité croissante, le développement des systèmes embarqués et temps réel nécessitent conjointement l'application de principes de génie logiciel et l'application de techniques formelles. Le travail développé pendant cette thèse propose une approche et des outils basés sur les modèles. Ces modèles, basés sur UML (Unified Modeling Language), permettent de définir un style architectural appelé SAIA[4] dont l'objectif est le développement et la mise au point de systèmes temps réel en intégrant l'évolution et la variabilité des plateformes. On entend ici par plateforme les services de communication entre le système et son environnement physique, c'est-à-dire des opérations de lecture et d'écriture via les capteurs et les actionneurs.

Pour répondre à cet objectif, l'idée de SAIA est de séparer clairement le modèle de plateforme du modèle de l'application. À cette fin, SAIA propose l'introduction d'une plateforme de communication abstraite avec le processus. Cette plateforme abstraite est composée d'*entrées* et de *sorties* utiles pour effectuer le contrôle, mais indépendantes d'une technologie de capteurs/actionneurs particulière. L'application est développée en se basant sur les services fournis par la plateforme abstraite.La stabilité d'une application de contrôle et sa qualité de contrôle sont, entre autres, dépendantes des caractéristiques temporelles de la plateforme abstraite. Cette dernière est donc composée d'un ensemble de services ainsi que d'une description de ses caractéristiques temporelles (notées QoS pour *Quality of Service*). La description de la QoS de la plateforme abstraite reflète le comportement temporel, sous forme de ω-expression régulière de la plateforme abstraite pour laquelle l'application a le comportement souhaité. Ainsi, nous avons d'un côté un modèle de la plateforme abstraite et de la QoS permettant la correction de l'application et de l'autre un modèle de la plateforme réelle dont la QoS a été analysée. Afin de connecter la plateforme abstraite à la plateforme réelle, SAIA s'appuie sur un connecteur complexe. Ce connecteur complexe est un assemblage de composants, décrit formellement par des automates temporisés réalisant des services de formatage, d'interprétation, de fusion de données et enfin d'adaptation de la QoS. Le connecteur complexe possède un comportement et modifie donc la QoS de la plateforme réelle. Afin d'évaluer l'impact du connecteur complexe sur la QoS de la plateforme réelle, une analyse formelle basée sur la simulation exhaustive du connecteur complexe est réalisée. Il est alors nécessaire de s'assurer que cette QoS nouvellement évaluée satisfait la QoS de la plateforme abstraite et permet ainsi la réalisation d'un système correct. La vérification de cette satisfaction est basée sur l'établissement d'un contrat de QoS. Dans SAIA, l'établissement d'un contrat de QoS est basé sur une relation de satisfaction (équivalence de trace) entre systèmes à transitions étiquetés. Enfin, SAIA a été mis en œuvre à plusieurs reprises dont, lors de deux concours d'implémentation de robots d'exploration terrestre dans le cadre de workshop satellites de RTSS (Real Time System Symposium).

Abstract

Due to their growing complexity, real time and embedded systems development requires to apply both software engineering and formal methods. In order to improve reuse, applying software engineering allows separation of concerns to be applied between the application part (control) and its communication with the physical environment. This separation allows the control to be deployed through different concrete platforms in terms of sensors and actuators. SAIA[4] implements these concepts by defining an architectural style where the control is based on an abstract platform. Then, to realize the system, the abstract platform is linked to a concrete platform through a complex connector. This software engineering method is promising, however an important real time and embedded systems aspect is that their correctness depends on the communication with the physical environment temporal behaviors. A minor change in a measure acquisition by a sensor can lead to unpredictable reaction of the system. To allow separation of concerns to be correctly realized, SAIA proposes formal models and techniques based on timed automata and exhaustive simulation. In a first step, formal QoS (Quality of Services) requirements are expressed in both the abstract and the concrete platform. Then, the second step ensures the connection correctness between the concrete and the abstract platform. This is done when the complex connector establishes a QoS contract. In SAIA, a QoS contract defines a formal conformity relation to ensure the QoS satisfaction and, so, to ensure a safe deployment during the connection. At the end, a show case demonstrates the use and this approach interest.

[4] *Sensors Actuators Independent Architecture*