Architectures logicielles pour les systèmes embarqués temps réel

Jean-Philippe Babau, Julien Deantoni

▶ **To cite this version:**

Jean-Philippe Babau, Julien Deantoni. Architectures logicielles pour les systèmes embarqués temps réel. Ecole d'été temps réel, Sep 2007, Nantes, France. inria-00587164

HAL Id: inria-00587164
https://hal.inria.fr/inria-00587164

Submitted on 19 Apr 2011

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

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

Architectures logicielles pour les systèmes embarqués temps réel

Jean-Philippe Babau, Julien DeAntoni
CITI laboratory – INSA-Lyon
20, avenue Albert Einstein, 69621 Villeurbanne cedex
jean-philippe.babau ; julien.deantoni@insa-lyon.fr

Résumé

Cet article propose un état de l'art sur les architectures logicielles dans le domaine de l'embarqué et du temps réel. Il présente et discute des entités à mettre en œvre tels que le modèle à composant (composants, interfaces, connecteurs), du niveau de description des configurations (logique, support, opérationnel), des contraintes de structuration (styles architecturaux), des langages de description (ADL), des types d'analyse et des stratégies de mise en œuvre pour l'exécution. Puis, pour illustrer l'importance des contraintes de structuration, deux styles architecturaux sont présentés. Le premier, Qinna, est dédié à la description de politiques de gestion dynamique de contraintes de QoS. Puis après avoir discuté de manière générale des principes d'indépendance vis-à-vis des plateformes, le deuxième style, SAIA, se propose d'assurer l'indépendance des applications de contrôle de procédé vis-à-vis des entrées/sorties.

1 Introduction

Dès la fin des années 60, Dijkstra expose certains principes de structuration [28] comme l'organisation en couche hiérarchique et l'abstraction permettant de faciliter le développement et la validation de systèmes informatiques. Toutefois, ce n'est que dix ans plus tard que la complexité des systèmes logiciels a fait émerger l'idée de conception architecturale [26]. Cette dernière 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 [43].

Concevoir une architecture logicielle aboutit à décrire un système comme une assemblage de composants. Raisonner au niveau architectural permet ainsi de maîtriser l'organisation du système par une décomposition en sous-systèmes ou composants. Une fois les composants spécifiés et validés, construire un système revient alors à réaliser un assemblage de ces composants, soit une problématique de composition.

Les bénéfices attendus par l'application d'une telle approche sont nombreux, tant au niveau de la conception (séparation des préoccupations), que de la qualité du logiciel produit (réutilisation, maintenance, évolution), ou encore de l'analyse de propriétés. Dans le domaine des systèmes embarqués temps réels, l'utilisation des ressources et l'aspect temporel forment alors un point essentiel des principes de composition et des outils d'analyse.

Après une discussion sur les définitions et les bénéfices attendus de l'utilisation des architectures logicielles, l'article fait un tour d'horizon des concepts à mettre en œuvre pour décrire une architecture logicielle dans le domaine des systèmes embarqués temps réel, soient le modèle à composant (composants, interfaces et connecteurs), le langage et le niveau de description des entités manipulées et enfin les contraintes d'organisation de ces entités. Après avoir discuté des techniques d'analyse et des principes de mise en œuvre pour l'exécution, les contraintes d'organisation étant primordiales pour obtenir les propriétés recherchées, nous présentons deux styles architecturaux. Le premier, Qinna, traite de la gestion dynamique de contraintes de QoS liées à l'utilisation de ressources limitées. Puis après avoir abordé la problématique de l'indépendance vis-à-vis de plateformes, le deuxième, SAIA, s'intéresse à l'indépendance des applications de contrôle de processus vis-à-vis des entrées/sorties possédant des contraintes temporelles.

2 Principes des architectures logicielles

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 cet article, deux définitions qui nous appa-

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

raissent suffisamment générales et pertinentes. Dans le standard 1471 [42] 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 [39] 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 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 ou autres (économiques,...) du système visé.

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 approche permet une décomposition du problème en sous problèmes. On assiste ainsi à une séparation des préoccupations essentielle pour la maîtrise de la complexité [5, 39].

La séparation des préoccupations fournit une première conception du système où certains détails sont masqués. Le concepteur peut ensuite 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 détail adapté à leur tâche [26, 43, 5, 39], et en proposant une interface claire et précise de chaque composant du système.

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 [45]. L'ensemble de ces contraintes, appelé style architectural, permet d'obtenir certaines propriétés de qualité au sens du génie logiciel, comme la réutilisation, la capacité d'analyse, l'interopérabilité, etc [69].

Enfin, associée à des techniques formelles, l'utilisation d'une architecture logicielle permet d'effectuer des analyses sur les diverses parties du système et sur leur composition [40, 39, 45]. Ces analyses 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 peuvent être faites à plusieurs stades du raffinement de l'architecture [4, 13].

On peut résumer les principaux bénéfices attendus lors de l'utilisation d'une architecture logicielle selon quatre aspects. 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.

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

2.3 Décrire une architecture

Aux vues des études existantes et en se basant sur la synthèse effectuée dans [65], une architecture est

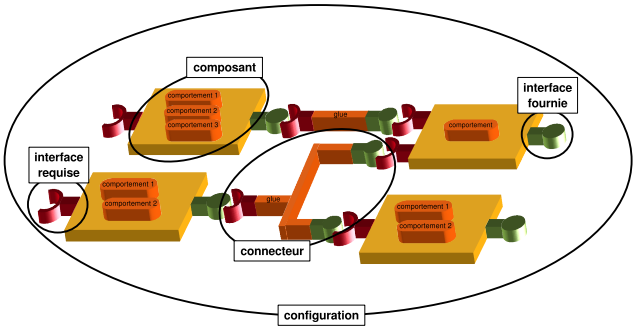

FIG. 1 – Représentation graphique d'une architecture logicielle

caractérisée par les éléments suivants (cf figure 1) :

- **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.

Par la suite, nous présentons et discutons chacun de ces concepts nécessaires à l'établissement d'une architecture logicielle, en particulier dans le domaine de l'embarqué et du temps réel. Nous discutons ensuite des possibilités d'analyse et de mise en œuvre des architectures logicielles.

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. A un composant est associé un comportement. On distingue alors deux manières de décrire le comportement d'un composant :

- 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, lui, spécifié de diverses manières. La littérature met en avant quatre catégories non exclusives :

1. Spécification du comportement par un (ou plusieurs) exécutable(s) binaire(s) ou sous forme de byte code (osgi [73], .Net [67], EJB [68], EAST-EEA [75], PACC [87]). 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 [83], PBO [79], rubus [48], PECOS [44], Think [32], MetaH [84], EAST-EEA [75], VEST [78], giotto [47], AADL [74]). Dans le contexte des systèmes embarqués, le langage C est alors un choix récurrent. Dans ce cas, on parle de composants en boîte blanche car le code source du composant est fourni et visible.
3. Spécification de paramètres et/ou de propriétés permettant de contrôler et/ou de caractériser le comportement du code. On retrouve ici des paramètres de configuration du code ainsi que le cycle de vie du composant. Dans le domaine de l'embarqué et du temps réel, les conditions initiales, le choix d'un algorithme ou une période d'activation sont des paramètres classiques de configuration. De son coté, le cycle de vie du composant est une propriété explicite des composants (cf. le modèle générique FRACTAL [11]). Dans le domaine visé, le cycle de vie contient des informations liées aux activités de déploiement et d'activation d'un composant, mais aussi des informations sur l'état du composant vis-à-vis de son exécution et de l'utilisation de ressources. Par exemple, EMPRESS [88], basé sur OSGI [73], se propose de définir pour chaque composant la façon selon laquelle le composant utilise ses ressources durant son cycle de vie. De son côté, Qinna [50], propose de spécifier un niveau d'utilisation des ressources selon les paramètres de configuration choisis.
4. Spécification du comportement à l'aide d'un langage formel. Cette approche permet de réaliser diverses analyses sur un 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 [29], MetaH [84], AADL [74], PECOS [44], COTRE [31], C2 [27], PACC [87]). Le comportement d'un composant peut être abs-

trait 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 [29], MetaH [84], EAST-EEA [75], VEST [78], giotto [47]). 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 quatre catégories précédentes ne sont pas exclusives, beaucoup d'approches associent différentes descriptions possibles d'un comportement. Typiquement, 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 [84], EAST-EEA [75], VEST [78], giotto [47], PACC [87], koala [83], PECOS [44]) ou nécessaires au déploiement (osgi [73], .Net [67]). La solution à retenir dépend 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, 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 est à noter 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 composant sur des cibles hétérogènes, un langage de programmation classique est préférable. Par contre, une phase de compilation doit être réalisée, phase dirigée selon le support utilisé (directives de compilation). De manière générale, l'installation d'un composant requiert des informations supplémentaires pour assurer un déploiement correct.

Déploiement Le déploiement correct d'un composant nécessite plusieurs informations supplémentaires au code. Celles-ci peuvent être générales, comme le nom de l'auteur ou le numéro de version ; être utiles à l'installation, comme le support d'exécution, 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. Pour un déploiement en-ligne, des outils tels OSGI [73] et CDDLM [89] pour l'embarqué, assurent la gestion dynamique du cycle de vie. Dans le cadre des systèmes temps réel fortement contraints, il est d'une part nécessaire de définir les ressources nécessaires à l'exécution du composant à un niveau de granularité plus fin et d'autre part, de vérifier, a priori, la correction du système après déploiement. Dans ce cadre, le lien entre un composant, son implémentation et les ressources nécessaires à son exécution est fait statiquement, a priori (VEST [78]).

Analyse Les deux premiers aspects se focalisent sur la problématique d'exécution d'un composant. De leur coté, les comportements abstraits sont utilisés pour s'assurer de la correction d'un composant primaire ou composite, soit lors de la connexion d'un composant à un autre composant, soit lors de son déploiement ou encore lors de son exécution.

Il faut noter que la plupart de ces propriétés sont liées au comportement du composant à l'exécution. Un des points importants et non trivial est alors de s'assurer que le comportement ainsi décrit de manière abstraite reflète le comportement à l'exécution du composant. A 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 [29], EAST-EEA [75]) ou le résultat de l'analyse de son exécution (MetaH [84], VEST [78]).

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.

2.3.2 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 [79], rubus [48], giotto [47]) ou des interfaces fournies ou requises (EAST-EEA [75], koala [83], Think [32], UML2.0 [72]). 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 aussi classées suivant le type d'interaction qu'elles implémentent. En se basant sur les approches existantes, on discerne trois principaux types d'interfaces :

- les interfaces fonctionnelles qui permettent d'accéder à un service particulier d'un composant ;
- les interfaces de configuration qui permettent d'ajuster certains paramètres relatifs au comportement d'un composant (conditions initiales, choix d'un algorithme, période d'activation, etc) ;

– les interfaces de synchronisation qui permettent de déclencher ou de suspendre l'exécution d'un composant (gestion du cycle de vie).

Puisque les interfaces sont les seuls points de communication d'un composant avec son environnement, elles représentent la spécification du composant. Idéalement, toutes les dépendances de contexte doivent être capturées par les interfaces du composant. Ainsi, un composant peut être composé et déployé indépendamment dans un système [80]. Afin d'atteindre cet objectif, [8] 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 [71] 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 [55].

Contrat de niveau 3 : il spécifie le protocole de synchronisation entre les appels aux différents services composant le comportement d'un (ou de) composant(s). Cette synchronisation peut être exprimée par des champs textuels comme une séquence ou de manière plus complexe par des automates spécifiques de type "*protocol statechart*" [60].

Contrat de niveau 4 : il spécifie les caractéristiques extra-fonctionnelles (sécurité, retard,...) que l'on désire observer sur le service associé. Il exprime ainsi le niveau de QoS (*Quality of Service*) requis ou offert pour/par une interface comme dans [50, 52]. A ce titre le contrat de niveau 4 est aussi appelé contrat de QoS. Dans le domaine de l'embarqué, un contrat de QoS correspond à une contrainte quantifiée et/ou temporisée sur un appel de service(s). Il est à noter que les approches intègrent rarement ce niveau de contrat dans leurs descriptions.

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 [80]).

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. Ainsi on s'assure des principes d'encapsulation pour les aspects fonctionnels via des contrats de niveau 1 et 2. Puis pour s'assurer d'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.

Il est à noter que 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 contrats 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 QoS est alors basée sur un opérateur de comparaison de quantité de ressources utilisées.

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.

2.3.3 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 2.3.2.

Certaines approches considèrent les connecteurs comme de simples liens logiques (MetaH [84], giotto [47], PECOS [44], PBO [79], rubus [48], koala [83]). 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 [76], Think [32], VEST [78], BIP [6]). Il a pour objectif l'adaptation des interfaces afin de permettre l'établissement de contrats entre des interfaces hétérogènes.

Un connecteur spécifie également la politique 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 [44], PBO [79], rubus [48]). 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 [29], Wright [2], BIP [6]).

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 [76], Think [32], VEST [78], soit de manière spécifique comme dans BIP [6]. La "glue" peut également être choisie parmi une liste finie de comportements (CLARA [29]).

Conclusion sur les connecteurs Dans la littérature, 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 liée au domaine d'application. Les connecteurs simples sont, de préférence, réservés aux systèmes ayant des besoins d'installation/remplacement de composants durant 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 terme de communication/synchronisation sont plus forts. En particulier, ils permettent de faire communiquer des composants hétérogènes en terme de signature (contrat de niveau 1), de comportement (contrat de niveau 2) et de protocole de synchronisation (contrat de niveau 3). A notre connaissance, à part dans SAIA [21], 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 pour réaliser un assemblage. Le résultat d'un assemblage est appelé configuration. La section suivante introduit cette notion.

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. A 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 reprenons ici la terminologie proposée par [29] :

Configuration ou architecture logique : c'est une vue architecturale des aspects fonctionnels d'un système sans prise en compte des détails d'implémentation. 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 [29] et EAST-EEA [75] 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 : c'est une vue architecturale du support d'exécution. 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 (service d'accès aux tâches, aux media de communication, etc.). A 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 [78], EAST-EEA [75]).

Configuration ou architecture opérationnelle : c'est une vue architecturale résultant du déploiement de l'architecture logique sur l'architecture technique. Elle décrit le système complet, avec les tâches, les protocoles de communication, la politique d'ordonnancement, etc. Il est alors possible de donner une description précise de l'utilisation des ressources. 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 approches se concentrent sur la description d'une configuration opérationnelle ou confondent la configuration logique et la configuration opérationnelle (MetaH [84], giotto [47], COTRE [7], PECOS [44], PACC [87], PBO [79], rubus [48], AADL [74]). Cette vue architecturale permet 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'or-

donnancement temps réel (MetaH [84], giotto [47], COTRE [31], rubus [48]), l'évaluation des temps de réponse (MetaH [84], COTRE [31]) ou le calcul de l'empreinte mémoire (Koala [83]).

On trouve cependant quelques approches qui identifient clairement la configuration logique (CLARA [29], EAST-EEA [75], VEST [78], koala [83]). 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 [33] , EAST-EEA [75]).

Il nous apparaît intéressant de séparer la configuration logique de la configuration opérationnelle afin de séparer les préoccupations liées aux fonctionnalités attendues des préoccupations d'exécution. Dans le domaine visé, ce découpage n'a de sens que si on considère des contraintes de QoS au niveau logique. Ce travail de dérivation de contraintes de QoS au niveau des composants reste éminemment complexe. Il doit s'appuyer sur des compétences métier fortes en terme d'analyse de performance. Au final, construire une architecture correcte revient alors à s'assurer de la correction de l'architecture logique ; puis à s'assurer de la faisabilité de la dérivation proposée 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.

2.3.5 Le style architectural

Les contraintes permettant d'obtenir des 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*" [35].

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 [41], 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*" [77]. Il en existe bien sûr d'autres et chacun de ces styles possède des variantes. De plus certains styles sont le résultat d'une combinaison 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 [45]. [57, 58] 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 recherchées : 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*) [5]. De nombreuses études, parmi lesquelles [81, 54, 53], se focalisent sur leur évaluation.

Selon les propriétés recherchées, un style architectural est plus ou moins efficace [18]. Il est possible de modérer 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. 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* [77].

Pour leur part, les propriétés chiffrables intègrent les aspects temporels et de performance. Certains styles architecturaux, en imposant un langage formel et une structure particulière facilite l'évaluation de ces propriétés [41]. Un style architectural adapté doit donc définir des types de composants et de connecteurs dont le comportement permet l'analyse de la propriété recherchée. Par exemple, 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 : pour faire une analyse RMA [56], on ne doit spécifier que des tâches périodiques ou sporadiques dont la pire durée est connue. Il faut donc noter qu'un style architectural impose la spécification des éléments nécessaires pour effectuer une analyse de propriétés chiffrables 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 la construction de configurations analysables et donc validables.

Pour autant, comme le souligne [35], il n'existe pas de style architectural "miracle" (*"silver-bullet"*). Chaque style possédant ses avantages et ses inconvénients, c'est à l'architecte de faire ou d'appliquer le style favorable à ses objectifs.

Il faut faire une différence franche entre l'évaluation de propriétés extra-fonctionnelles non chiffrables liées à un style architectural et la propension qu'apporte un style architectural à l'évaluation de propriétés extra-fonctionnelles chiffrables. On peut par exemple dire qu'un style architectural 'X' permet d'obtenir un bon niveau de 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 de description d'une architecture (composants, connecteurs, etc) sont intimement liés au modèle à composant sous-jacent. A l'inverse, un modèle à composant, avec des contraintes de composition fortes, peut définir implicitement un style architectural [19, 86]. Pour autant, si ce style architectural implicite est trop contraint, le modèle à composant correspondant n'est a priori adapté qu'à un type de problème ou à un domaine spécifique. Il y a alors confusion entre le modèle à composants et le style architectural. De notre point de vue, un style architectural doit être indépendant d'un modèle à composant spécifique. Nous préconisons donc de le définir vis-à-vis d'un modèle de composants minimal. Ce dernier représente l'ensemble des contraintes sur les modèles à composants pouvant accepter l'application du style.

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.

2.3.6 Les langages

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 [84], giotto [47], COTRE [7] , AADL [74], CLARA [29], EAST-EEA [75], VEST [78], unicon [76], Wright [2]) et les modèles à composants (notés CMs pour *Component Model*) (PECOS [44], PACC [87], PBO [79], rubus [48], koala [83], Think [32], osgi [73] , .Net [67], EJB [68]).

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 [72] est une solution extensible et, de fait, standardisée qui permet la description de composants, d'interfaces et de connecteurs, soient les éléments nécessaires pour décrire des architectures. Puisque les entités décrites dans UML2 sont génériques, certains langages de description d'architecture comme AADL [74] proposent une spécialisation des entités d'UML pour définir leur langage.

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 d'outils de transformations de modèles.

Enfin, il est à 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 [41] 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*) [71] ou par la définition d'un méta-modèle [37].

Conclusion sur les langages 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 developper indépendemment les un des autres.

Au niveau des formalismes, on voit de plus en plus d'ADL basés sur UML (AADL [74], EAST-EEA [75], VEST [78]) ; UML ayant l'avantage de fournir une notation standardisée de plus en plus utilisée et des outils existants. Appuyé par l'initiative MDA [70], l'utilisation de modèles pour la description d'une archi-

tecture est, aujourd'hui, l'approche qui nous semble la plus appropriée. Elle permet de définir clairement les concepts d'un domaine et de proposer une base pour établir des liens entre les domaines.

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 [64] 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. Ces analyses sont le sujet du paragraphe suivant.

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 [50], osgi [73], .Net [67], EJB [68]). A l'inverse, les systèmes considérés ici 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 2.3.5, il existe deux grandes catégories de propriétés vérifiables. La première concerne les propriétés non chiffrable comme le niveau de réutilisabilité de l'architecture ou sa facilité de maintenance. Les études de ces propriétés, parmi lesquelles [81, 54, 53], sont basées sur des techniques non formelles et non automatisables (cf. analyse de [85]).

La deuxième catégorie concerne les propriétés chiffrables comme les propriétés temporelles, de performances, etc. Ces propriétés sont essentielles à la validation des systèmes embarqués temps réel. Les travaux décrits dans [20] proposent une classification de ces propriétés selon deux classes :

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*) ou [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 de vivacité (quelque chose de bien arrivera un jour), de sûreté (quelque chose de mauvais n'arrive jamais) et les propriétés temps réel (le temps écoulé entre deux événements dans le système est inférieur ou égal à un temps donné) (COTRE [31], CLARA [33], EAST-EEA [75], MetaH [84]).

Ces analyses, ainsi que celles réalisées dans [63, 85], se basent sur une description formelle et abstraite du comportement d'un composant à l'aide d'automates temporisés. L'évaluation est alors réalisée par des outils basés sur les techniques formelles tels que UPPAAL [61], Kronos [90] ou CADP [34].

On trouve également, parmi les propriétés dérivées, les analyses d'ordonnançabilité (MetaH [84], giotto [47], PACC [87], COTRE [31], rubus [48], VEST [78], Unicon [76]). 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 politiques de 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 analysé unité par unité pour chaque configuration partielle 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 [83]. Ces propriétés peuvent être vérifiées lors de chaque connexion d'un composant avec le système, soit au fur et à mesure de la construction d'une configuration. Elles 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 [71], d'un méta-modèle [37] ou d'un artefact spécifique à l'ADL comme dans [41, 2].

Conclusion sur les analyses

Les analyses de propriétés non chiffrables servent uniquement de base de comparaison entre différentes configurations candidates. Elles sont indépendantes d'un ADL particulier. Les méthodes d'analyses 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 chiffrables 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 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.

2.5 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. A ce niveau, on distingue principalement deux stratégies :

- la projection du code exécutable de chaque composant et de chaque connecteur directement sur une infrastructure permettant son exécution (cf. figure 2) ;
- la compilation de la configuration vers un code exécutable et monolithique de la configuration (cf. figure 3).

Dans la première stratégie, l'exécution des composants est gérée par un intergiciel spécifique permettant l'installation et l'exécution du code exécutable de chaque entité architecturale (via les fabriques, les usines à liaison, des outils pour l'établissement des contrats ou la gestion du cycle de vie). 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. 2.3.3) 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 B). Lorsqu'au contraire les connecteurs sont des entités complexes, le code exécutable de la "glue" du connecteur est exécuté par l'intergiciel, comme pour un composant (cf. figure 2 A).

Dans la deuxième stratégie, la configuration est compilée dans un langage intermédiaire compilable/exécutable sur le support d'exécution (configuration technique). Les composants ne sont plus réifiables à l'exécution ; le découpage en composants de la vue architecturale disparait à l'exécution.

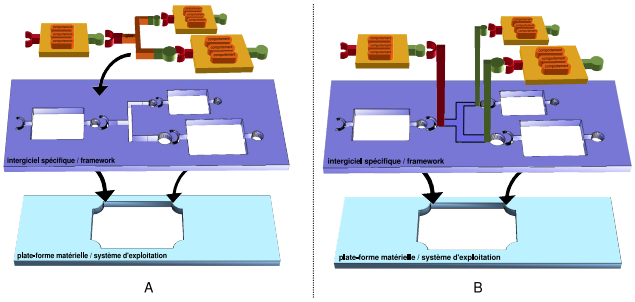

FIG. 2 – Deux mises en œuvre d'une configuration basée sur un intergiciel

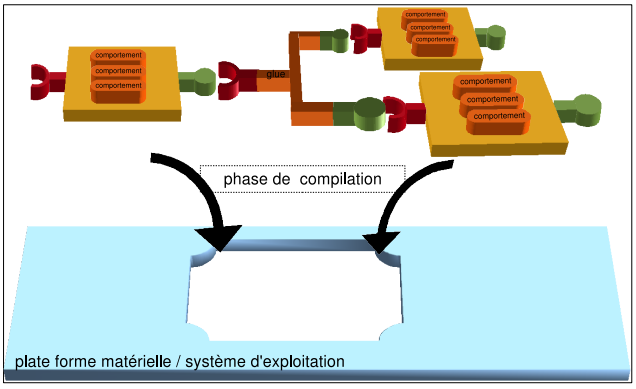

FIG. 3 – l'implémentation d'une architecture logicielle compilée

Conclusion sur la mise en œuvre d'une configuration

Le choix d'une stratégie est fortement lié à la stratégie choisie lors de la description du comportement des composants. Dans le cas d'une description du comportement sous forme de code source (formel ou non), le choix reste libre, même si généralement, dans le domaine de l'embarqué, la configuration est compilée. Ce choix est important et intervient très tôt dans le processus de 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 pré-établi et non optimisé (par exemple une tâche par composant).

La compilation d'une configuration permet d'obtenir un code plus efficace en terme d'empreinte mémoire et plus performant pour l'allocation des ressources. La structure de la configuration logique peut même être orthogonale au modèle de tâche choisi et donc à la structure de 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 [46, 38]. C'est

pour ces raisons que la compilation d'une configuration est la solution la plus fréquemment retenue dans le domaine des systèmes embarqués temps réel.

2.6 Conclusion sur la mise en place d'architectures logicielles

Dans les chapitres précédents, 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 est accessible au travers d'**interfaces**. Enfin les interfaces des composants sont interconnectées à l'aide de **connecteurs**. Un **ADL** est un langage permettant de représenter chacun de ces éléments.

Nous avons vu qu'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 se focaliser sur 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 sur-couche 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 généralement 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 QoS est nécessaire à l'utilisation de composants dans le domaine d'application visé où les contraintes de QoS sont fortes, voire critiques.

Au niveau de la mise en place des connecteurs, une première approche consiste à utiliser des connecteurs logiques. Dans ce cas, les interfaces ainsi connectées se doivent d'être homogènes. A 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 pour assurer l'établissement d'un contrat.

Une attention particulière est apportée à l'analyse de tels systèmes. Ces analyses concernent principalement l'utilisation des ressources (empreinte mémoire, ordonnancement, etc.) et les contraintes essentiellement temporelles (échéance de bout en bout, etc.). On distingue alors deux types d'analyses :

- les analyses réalisées sur une configuration : l'utilisation de ce type d'analyse est souvent réservé à l'évaluation de propriétés émergentes d'un système. Elles nécessitent une vue boîte grise du comportement des composants et des connecteurs. Il faut noter que ce comportement, en particulier au niveau temporel, doit refléter le comportement des composants à l'exécution.
- l'établissement d'un contrat de QoS lors de la connexion entre deux interfaces : c'est le connecteur qui doit s'assurer de l'établissement de ce contrat. On s'assure ainsi d'une composition correcte vis-à-vis de contraintes directement composables. La spécification des propriétés requises et fournies décrites par les interfaces doit être homogène et non ambigüe. De plus, la relation de satisfaction qui spécifie les conditions sous lesquelles le contrat peut être établi, doivent être formellement spécifiées.

Les approches décrites dans la littérature sont souvent homogènes au niveau du type d'analyse présenté précédemment, et pour la QoS, s'appuient généralement sur des analyses de configuration.

Dans tous les cas, comme souligné dans [40], 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, de par les contraintes de structuration et une sémantique formelle, est alors double :

- il permet l'application de principes de génie logiciel afin de posséder des qualités (au sens du génie logiciel) telles que la réutilisabilité.
- il permet de produire une configuration analysable du point de vue de propriétés chiffrables.

On voit ainsi émerger plusieurs aspects importants pour 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 de QoS 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 temporelle formelle. Enfin, un style architectural, en précisant les types de composants et les contraintes structurelles, permet d'obtenir un système analysable, mais aussi certaines propriétés de génie logiciel telles que la réutilisabilité.

Il est à noter qu'au vu de la complexité de mise en en œuvre d'une architecture logicielle, de nombreux travaux proposent des librairies de composants directement réutilisables. Ainsi, on facilite le travail du concepteur en lui fournissant des composants sur étagère qu'il lui reste à configurer et à assembler. On peut ici citer les connecteurs standards de communication de [29], la librairie KORTEX de Think ([82]). La mise en place de librairies pour un domaine peut apparaître comme l'objectif de l'architecte vis-à-vis d'un concepteur, dont le rôle devrait être, au final, de se limiter à intégrer les spécificités de son application.

De manière générale, des outils doivent accompagner le concepteur pour maîtriser la complexité et automatiser la gestion des modèles et des transformations sous-jacentes.

Maintenant que les principes pour la mise en place d'une architecture logicielle ont été étudiés, la partie suivante présente deux styles architecturaux, appelés Qinna et SAIA.

3 Styles architecturaux pour les systèmes embarqués temps réel

Nous illustrons maintenant l'intérêt d'un style architectural en considérant des aspects liés aux contraintes de QoS. Le premier style proposé, appelé Qinna [50], traite de la gestion dynamique de contraintes de QoS liées à l'utilisation de ressources limitées. Le deuxième, appelé SAIA [21], s'intéresse à l'indépendance des applications de contrôle de processus vis-à-vis d'entrées/sorties possédant des contraintes temporelles.

3.1 Le style architectural Qinna

3.1.1 Introduction à Qinna

Ce chapitre présente le style architectural Qinna qui propose un cadre pour la description de politiques dynamiques de gestion de la QoS liée à l'utilisation de ressources limitées. Qinna s'appuie sur le modèle à composants générique Fractal [11]. Les expérimentations ont été réalisées à l'aide de Think [32] qui est une implémentation de Fractal à base de langage C pour l'embarqué. Nous présentons ici les principes généraux du style architectural avant de discuter de son champ d'application.

3.1.2 Principes de Qinna

Qinna intègre les principaux concepts de gestion de la QoS définis par [12] au travers d'activité de gestion de contrats de QoS (spécification, initialisation et suivi). Un contrat de QoS représente un niveau de QoS contractualisé, après négociation, entre un composant requérant un service et un composant offrant le service correspondant. Pour décrire une politique de gestion dynamique de contrats de QoS, Qinna propose alors cinq types génériques de composant : le QoSComponent, le QoSComponentBroker, le QoSComponentManager, le QoSObserver et le QoSDomain. Les types et les services associés à ces composants sont eux aussi génériques. Ce cadre générique n'a pour objectif que de permettre l'expression de politiques dynamiques de gestion de la QoS, en conséquence Qinna n'a pas vocation à spécifier **une** politique de gestion de la QoS.

Nous détaillons maintenant le modèle de QoS et les types de composants définis par Qinna, avant d'aborder le comportement induit par Qinna.

QoS La spécification de la QoS, offerte ou requise, s'appuie sur un type abstrait possédant les opérateurs ($+$, -, $<$ et $==$) et une plus petite valeur notée *Zero* doit exister. Selon ce modèle, dans le domaine visé, on peut considérer des contraintes relatives à la quantité de ressources utilisées (la quantité mémoire, le débit réseau, le taux d'utilisation du CPU), au nombre d'invocation de services (nombre de création de tâches, ...) et enfin au niveau de satisfaction d'un utilisateur (par exemple bon, moyen et mauvais).

QoSComponent Un composant de type QoSComponent fournit, au moins, une interface fonctionnelle pour laquelle on souhaite gérer une contrainte de QoS. Afin de fournir un niveau de QoS sur ses interfaces offertes, un QoSComponent requiert un certain niveau de QoS sur ses interfaces requises et doit être configurable via une contrainte locale notée `T_CL`. La contrainte locale implique alors un niveau de QoS auquel le composant s'exécute.

QoSComponentBroker Un composant de type ComponentBroker se base sur une contrainte globale, notée `T_CG`, afin de pouvoir fixer, ou pas, la contrainte locale d'un QoSComponent. Le QoSComponentBroker est responsable du test d'admission (`T_CG_current` + `T_CL_required` $<$ `T_CG`) et de la réservation des QoSComponent (si le test d'admission

est validé, le QoSComponentBroker retourne une référence vers un QoSComponent). Il existe un QoSComponentBroker pour chaque classe de QoSComponentBroker.

QoSComponentManager Un composant de type QoSComponentManager encapsule une vue orientée QoS d'un composant de type QoSComponent. Il explicite pour un service fourni à un certain niveau de QoS, les niveaux de QoS requis sur les services requis. Ce lien s'exprime sous la forme d'une table ordonnée, dite table de mapping, qui est une liste ordonnée d'informations de liens : QoSofferte - QoSRequise. La relation d'ordre est basée sur le niveau de QoS fournie.

De plus, le QoSComponentManager implémente les mécanismes permettant l'adaptation dynamique de la QoS (recherche d'un niveau maximal possible, au sens de la QoS fournie). Il existe autant de QoSComponentManager que de QoSComponentBroker.

QoSDomain Un composant de type QoSDomain est le composant de plus haut niveau de l'architecture. Il définit un point d'entrée unique de l'architecture et encapsule tous les composants de l'architecture. A partir des niveaux d'importance (de type `entier`) définis par l'utilisateur, le QoSDomain est en charge des politiques d'adaptation (choix du contrat à dégrader/améliorer).

QoSComponentObserver Un composant de type QoSComponentObserver a pour rôle d'observer et de notifier des niveaux de QoS fournis par les interfaces fonctionnelles des QoSComponent lors de leur exécution. Il implémente donc les politiques d'observation des contrats. Il existe un QoSComponentObserver par classe de QoSComponent. En pratique, du fait de la complexité de l'observation, seuls les niveaux de QoS fournis par les ressources sont observés, ce qui revient à instrumenter chaque appel à la ressource pour contrôler son utilisation selon le contrat spécifié.

Comportement Les principales opérations considérées par Qinna sont la mise en place d'un contrat, le suivi des contrats et la mise à jour des données relatives aux contrats. Ces opérations sont menées de manière atomique et prioritaire afin de préserver la cohérence globale du système. Les opérations liées à l'établissement de contrats sont activées lors de chaque changement dans l'état du système. La mise en place d'un contrat est liée à un changement dans la structure par l'activation d'un nouveau composant. La mise à jour correspond aux adaptations dynamiques des contrats actifs suite à l'arrêt (annulation d'un contrat existant) ou à l'activation d'un nouveau composant, ou encore à un changement de l'état d'une ressource (modification des contraintes globales `T_CG`). Enfin, la mise à jour des données est réalisée lors de l'initialisation du système, via les tables de mapping.

De par sa structure distribuée, Qinna gère implicitement des contrats globaux de QoS. En effet, si un service d'un QoSComponent défini un arbre implicite d'appel de QoSComponent, un contrat est composé d'un arbre implicite de contrats dont les nœuds sont les lignes dans la table de mapping des QoSManager correspondants. L'établissement d'un contrat correspond alors à un parcours de l'arbre des sous-contrats. Qinna ne fournit pas d'élément sur l'algorithme de parcours de l'arbre, ni sur les politiques ou les mécanismes d'adaptation (ordre et principes de dégradation/amélioration).

3.1.3 Conclusion sur Qinna

Ces diverses contraintes de représentation définissent de fait un style architectural pour une gestion dynamique de contrats de QoS discrétisés pour les systèmes embarqués. L'utilisation de Qinna permet de :

- respecter les principes de séparation de préoccupation entre les aspects fonctionnels et non fonctionnels ;
- respecter les principes de séparation des préoccupations entre les politiques (QoSDomain) et les mécanismes (QoSManager) d'adaptation : le composant QoSDomain a la charge d'évaluer les niveaux d'importance des requêtes afin de demander des adaptions aux QoSComponentManagers ;
- d'intégrer des QoS hétérogènes (nous avons fait des expérimentations en ce sens) ;
- d'intégrer des politiques dynamiques et distribuées (au niveau des QoSManager) de gestion de contrats de QoS (parcours de l'arbre) ;
- et enfin de bénéficier des apports des composants en termes de réutilisabilité et maintenabilité.

Qinna identifie principalement deux domaines d'applications privilégiés que sont les systèmes multimédia et les systèmes temps réel. Dans le cas des systèmes temps réel, Qinna permet de structurer clairement les différents éléments de QoS en identifiant, en particulier, les éléments à mettre en œvre pour la gestion de l'hétérogénéité (temps réel dur et relâché, tâches périodiques et apériodiques [62]) et pour la gestion des modes dégradés des applications temps réel (par exemple, lors de la mise en œvre du régisseur temps réel [25]) [49]. Dans le cas des systèmes multimédia, Qinna apporte en plus les bénéfices liés à la gestion dynamique de la QoS [51].

Du fait que Qinna ne soit qu'un style architectural, son efficacité dépend grandement de son implémentation, soit de la granularité des composants, de la construction des tables de mapping et des algorithmes de parcours de l'arbre. Dans ce sens, nous

travaillons actuellement à un outil formel d'évaluation d'une architecture respectant les principes définis par Qinna pour un système embarqué (taux d'utilisation des ressources, nombre d'adaptations). De même, nous travaillons à fournir des librairies de composants Qinna pour des cas type tels que des gestionnaires de ressources (ordonnanceur de CPU, gestionnaire mémoire ou réseau, vérification du nombre d'instances activées).

Après avoir parlé de gestion dynamique de la QoS, nous abordons la problématique de l'organisation du logiciel pour permettre une indépendance vis-à-vis des plateformes d'exécution sous-jacentes.

3.2 Indépendance vis-à-vis des plateformes

3.2.1 Définitions

La réutilisation de logiciel 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 [36]. Une des tendances actuelles 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]. La notion de plateforme ne connaît pas à ce jour de définition précise et communément admise. MDA [70] 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 ses 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. Dans le domaine de l'embarqué et du temps réel, les plateformes sont classiquement des services d'exécution (architecture matérielle et exécutifs) et des services de communication (pilotes de communication avec le procédé et services de communication distante). De même, les intergiciels qui assurent l'exécution et la communication des composants est à situer dans cette catégorie.

Par la suite, nous conservons l'idée qu'une plateforme est un ensemble de services qui, couplés à une une description donnée, assure son exécution. Nous présentons maintenant rapidement les styles architecturaux pour assurer l'indépendance vis-à-vis des plateformes.

3.2.2 Styles architecturaux pour l'indépendance

L'étude des styles architecturaux dans le domaine des réseaux montrent 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 ([91, 14, 15]). Pour les IHM, la littérature identifie trois couches : l'interface avec l'application, l'interface avec l'environnement et la couche de liaison entre les deux précédentes [59],[16],[17]. 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, [1] ne propose aucune gestion des propriétés de QoS. D'autre part, 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 [66]. Cette approche, orientée par la technologie, n'admet pas de variabilités sur les fonctionnalités offertes par les périphériques réels. Pour admettre des variabilités fortes sur les plateformes, on préconise plutôt la mise en place d'une plateforme abstraite.

3.2.3 Plateforme abstraite

Pour assurer l'indépendance vis-à-vis d'une plateforme, [3] préconise l'introduction d'une plateforme abstraite. Une plateforme abstraite définit un ensemble de plateformes possibles qui sont pertinentes pour l'application. Dans les domaines visés, cette plateforme abstraite doit posséder des caractéristiques de QoS, représentant l'ensemble des caractéristiques possibles acceptables au niveau de la plateforme finale utilisée. Il existe deux stratégies pour modéliser une telle plateforme. Dans la première, l'ensemble des plateformes réelles a été prévu à l'avance. La plateforme abstraite représente alors autant d'implémentations que de plateformes réelles. Dans la deuxième stratégie, la plateforme abstraite est indépendante de toute plateforme réelle : elle représente une vue unifiée d'un ensemble de plateformes réelles. Une adaptation est ensuite nécessaire pour lier les deux vues. On trouve à ce niveau des solutions basées sur des relations de conformité (la plateforme réelle remplace la plateforme abstraite), sur la mise en place de connecteurs entre les deux (conservation des plateformes

abstraites et réelles et mise en place d'un lien entre les deux), voire sur des politiques de négociation (configuration de paramètres pour assurer la correspondance entre les deux).

Il apparaît indispensable de préciser la politique choisie et de s'assurer de sa correction. Cela revient à formaliser, en particulier au niveau de la QoS, le lien entre la plateforme abstraite et la plateforme réelle et à établir une relation de satisfaction entre les deux.

Nous illustrons cet aspect avec SAIA qui permet d'assurer l'indépendance vis-à-vis de capteurs et d'actionneurs pour des applications de contrôle de procédé avec contraintes temporelles.

3.3 Le style architectural SAIA

3.3.1 Principes généraux de SAIA

L'approche SAIA a pour objectif le développement et la mise au point de systèmes de contrôle de procédé en intégrant l'évolution et la variabilité des plateformes. On entend ici par plateforme les services de communication avec le procédé, soit des opérations de lecture et d'écriture via les capteurs et les actionneurs. En intégrant la variabilité des plateformes, SAIA se propose de couvrir deux objectifs :

- une application développée et validée sur une plateforme simulée peut être déployée sur une plateforme réelle sans modification, ni ajustement ;
- une même application peut être déployée sur des plateformes différentes.

Pour répondre à ces objectifs, l'idée de SAIA est de séparer clairement le modèle de la plateforme réelle du modèle de contrôle. A cette fin, SAIA propose l'introduction d'un modèle de la plateforme de communication abstraite avec le procédé, nommé SAIM (*Sensors/Actuators Independent Architecture*). Cette plateforme abstraite est composée d'*entrées* et de *sorties*, utiles pour effectuer le contrôle, mais indépendantes d'une technologie de capteurs/actionneurs. Par exemple la vitesse mesurée d'un robot est une *entrée* nécessaire à l'application de contrôle mais est indépendante du type de capteur utilisé (GPS, codeur incrémental) et du mode d'acquisition de la vitesse. Afin de connecter la plateforme abstraite à la plateforme réelle, cette dernière étant modélisée dans le SAM (*Sensors Actuators Model*), SAIA s'appuie sur un connecteur complexe, spécifié dans l'ALM (*Adaptation Layer Model*). Ce connecteur est un composant composite composé d'un assemblage de sous-connecteurs, un sous connecteur étant dédié à un type d'*entrée* ou de *sortie*. Les sous-connecteurs sont en charge de connecter des interfaces hétérogènes selon trois aspects : adaptation des unités via des services de formatage, interprétation des informations (transformation et/ou fusion de données) et enfin adaptation de la qualité de service. Ce dernier point est fondamental. En effet, les lois de contrôle sont très sensibles à la QoS des pateformes en termes de fréquence, de retard, de précision et de pertes.

3.3.2 SAIA et la QoS

Une application de contrôle de procédé est sensible à plusieurs paramètres qui influencent sa qualité du contrôle. La qualité de contrôle est un ensemble de contraintes exprimées sur le procédé en termes de stabilité ou de marges d'erreurs. Elle est, en particulier fortement liée aux caractéristiques temporelles de la plateforme. SAIA propose donc de dériver les contraintes de qualité de contrôle en contraintes temporelles sur la plateforme abstraite.

Une fois le SAIM et ses contraintes de QoS spécifiées, il faut être capable de connecter la plateforme abstraite à une plateforme réelle en s'assurant que les contraintes de QoS sont satisfaites. Pour ce faire, il est nécessaire que la plateforme réelle inclue une description de ses propres contraintes de QoS. La correction de la connexion entre la plateforme abstraite et la plateforme réelle est alors vérifié par l'établissement d'un contrat de QoS entre les contraintes requises et les contraintes fournies.

3.3.3 Le contrat de QoS dans SAIA

Dans SAIA, l'établissement d'un contrat de QoS est réalisé 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. La QoS est ici un ensemble de caractéristiques temporelles, comprenant la loi d'arrivée et la loi de retard et la loi de corrélation temporelle, caractérisant les occurrences d'un flot d'information.

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 interfaces et donc les informations circulant entre le SAM et le SAIM. En modifiant ces informations, la "glue" modifie également leur QoS. Avant de pouvoir établir le contrat de QoS, l'impact de la "glue" sur la QoS est évalué par la réalisation d'analyses temporelles exhaustives. Dans ce but, le comportement de chacun des composants de SAIA est décrit à base d'automates temporisés communicant IF [9]. Il est alors possible de réaliser une simulation exhaustive du système aboutissant à un Système à Transitions Étiquetées (STE.). Ce STE contient tous les chemins d'exécution du connecteur complexe. De plus, cette exécution peut être instrumentée, via des observateurs, afin d'évaluer la QoS en sortie de chacun des sous connecteurs. La QoS modifiée par les sous-connecteurs est exprimée sous forme d'un STE représentant la loi des arrivées, des retards

et de la cohérence temporelle lors de fusion de données.

Il est ainsi possible de vérifier si la connexion entre le SAM et le SAIM donne lieu à l'établissement d'un contrat ou non (cf. figure 4).

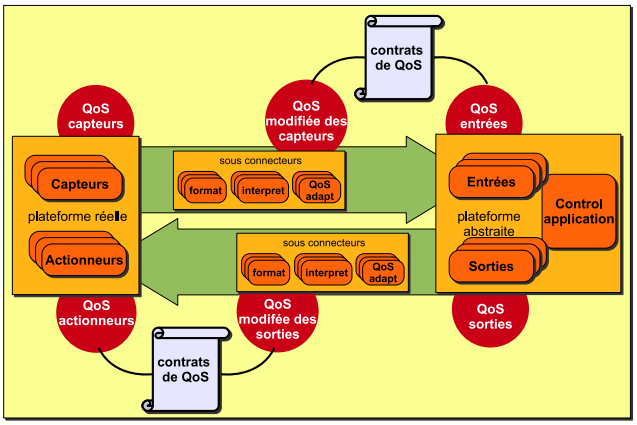

FIG. 4 – Gestion de la QoS dans SAIA

L'établissement d'un contrat de QoS s'appuie sur une relation de satisfaction dépendante de la manière d'exprimer les contraintes de QoS. Les contraintes peuvent être exprimées de plusieurs manières, plus ou moins expressives : une constante, un intervalle ou un langage exprimé sous forme d'ω-expressions régulières. On obtient ainsi trois relations de satisfactions différentes : l'égalité, la relation d'inclusion et la relation de sous langage respectivement pour les constantes, les intervalles et les ω-expressions régulières. Dans ce dernier cas, la relation de sous langage est réalisée sur les STE via l'application d'une relation d'équivalence de trace [10]. Au final, si pour chaque caractéristique, un contrat de QoS peut être établi, on considère la connexion correcte.

L'approche proposée offre une formalisation de la correction d'un assemblage donné par l'établissement de contrats de QoS. On a présenté les principes formels permettant d'assurer la correction d'un assemblage donné, on présente maintenant les expérimentations réalisées.

Au niveau de la mise en œuvre d'une configuration, SAIA ne présume pas d'une stratégie par compilation ou basée sur un intergiciel. En pratique, les expérimentations s'appuient sur une approche par projection du modèle sur des tâches gérées par un exécutif multitâches.

3.3.4 Expérimentations de SAIA

SAIA, dont un outil de modélisation est disponible [24], a été expérimenté vis-à-vis des objectifs attendus, sur plusieurs études de cas. La première concerne le développement du contrôle d'un robot d'exploration pour un concours proposé en temps qu'événement satellite de la conférence RTSS'05 (*Real Time System Symposium*). Son implémentation a permis de d'illustrer l'approche et de mettre en avant la réalité exécutable des modèles proposés. De plus, puisque que le concours imposait l'utilisation d'une plateforme de communication simulée, les avantages de SAIA lors du déploiement sur une plateforme de communication réelle ont été mis en avant [22]. Afin de mettre en avant les qualités de réutilisation de l'application de contrôle sur différentes plateformes de communication, nous avons participé au concours similaire l'année suivante [23]. lors de ce nouveau concours, la plateforme de communication était complètement différente de celle du premier concours, cependant les objectifs du contrôle restaient proches. Le modèle SAIM du premier concours ainsi que son implémentation ont donc été réutilisés et étendus. Il a ainsi été possible de fournir les modèles et les implémentations pour le deuxième concours sans modification de l'application de contrôle. Ces mises en œuvre ont ainsi permis de montrer la propension du style architectural SAIA à répondre au problème de déploiement correct d'une application de contrôle sur plusieurs plateformes de communication.

3.3.5 Conclusion sur SAIA

SAIA propose la structuration d'une architecture logique selon deux couches : la plateforme abstraite et la plateforme réelle. La première est indépendante des technologies d'acquisition/restitution sous-jacentes. Elle possède des contraintes temporelles exprimées sous forme d'un langage (constante, intervalle, expression régulière) qui expriment les contraintes temporelles requises pour assurer un fonctionnement correct de l'application de contrôle. En partant de l'hypothèse que les caractéristiques de QoS fournies par la plateforme réelle sont connues, SAIA propose la mise en place d'un connecteur complexe assurant l'adaptation des interfaces (formatage, interprétation, QoS) entre les deux couches. L'analyse de ce connecteur permet d'extraire ses propriétés temporelles et de vérifier ainsi la relation de satisfaction contractualisée entre les deux plateformes via ce même connecteur. Au final, SAIA fournit les outils pour la spécification et l'évaluation de contrats de QoS temporels.

Dans la même idée que SAIA, les travaux présentées dans [30] se propose de construire un module de communication avec l'environnement externe indépendant des technologies de communication (protocoles) sous-jacente. L'étude se limite aux technologies de communication sans fil orientées connexion (IrDA, Bluetooth). L'étude s'intéresse plus particulièrement à une description abstraite de la qualité de service et à la mise en place d'adaptation dynamique de la QoS entre la couche de communication abstraite et les plateformes réelles.

4 Conclusion

Mettre en place une architecture logicielle requiert de préciser le modèle à composant (composant, interface et connecteur), les contraintes de structuration (style architectural), le langage (ADL), le niveau de description visé pour les configurations, une sémantique opérationnelle et les techniques d'analyse associées, et enfin les principes et les outils de mise en œvre pour l'exécution.

Dans le domaine des systèmes embarqués temps réel, le comportement d'un composant logiciel est généralement décrit en langage C, puis compilé et exécuté sur un système d'exploitation temps réel. Dans la quasi totalité des approches, la description architecturale résulte alors 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 émergentes 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 tout en s'assurant du respect de contraintes de QoS. De même, les contrats de niveau 4 sont rarement traités.

Les styles architecturaux présentés, Qinna et SAIA, fournissent un cadre pour l'expression de politiques de gestion dynamique de contrats de QoS et permettent d'assurer l'indépendance des applications de contrôle vis-à-vis d'entrées/sortie avec contraintes temporelles. Ces études ont montré l'intérêt de posséder des modèles et des outils pour aider le concepteur à maîtriser la conception architecturale. En particulier, en assurant la séparation des préoccupations et en considérant des contraintes de QoS, les styles proposés fournissent des éléments d'aide au concepteur pour la maîtrise de la conception architecturale guidée par les performances (spécification de la QoS et analyse d'assemblages, formalisation de contrats de QoS).

Il existe bien sur de nombreuses perspectives à ces travaux comme assurer l'indépendance des applications vis-à-vis des services d'exécution multitâches et vis-à-vis de la distribution des calculs, formaliser le lien avec les architectures matérielles, intégrer des contraintes diverses (sécurité, ...). La résolution de ces problèmes devrait permettre, au final, la mise au point de services embarquables et adaptables, et la construction correcte de systèmes par assemblage de composants configurables.

Références

[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] R. 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] J. Almeida, R. Dijkman, M. van Sinderen, and L. 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] ARTIST. Adaptive real-time systems for quality of service management. *Draft IST-2001-34820*, 2003.

[5] L. Bass, P. Clements, and R. Kazman. *Software Architecture in Practice, Second Edition.* Addison-Wesley Professional, April 2003.

[6] A. Basu, M. Bozga, and J. 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.

[7] 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.

[8] A. Beugnard, J.-M. Jézéquel, N. Plouzeau, and D. Watkins. Making component contract aware. *IEEE computer, 32(7)*, pages 38–45, 1999.

[9] M. Bozga, S. 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.

[10] S. D. Brookes, C. A. R. Hoare, and A. W. Roscoe. A theory of communicating sequential processes. *J. ACM*, 31(3) :560–599, 1984.

[11] E. Bruneton, T. Coupaye, and J. Stefani. Recursive and dynamic software composition with sharing. In *7th ECOOP International Workshop on Component-Oriented Programming (WCOP'02)*, 2002.

[12] L. H. C. Aurrecoechea, A. T. Campbell. A survey of qos architectures. In *Multimedia Systems*, volume 6(3), pages 138–151, 1998.

[13] K. Chan. Formal proofs for qos-oriented transformations. *edocw*, 0 :41, 2006.

[14] J. Coutaz. Abstractions for user interface design. *Computer*, 18(9) :21–34, 1985.

[15] J. Coutaz. Abstractions for user interface toolkits. *Foundation for Human-Computer Communication, Elsevier Science, North-Holland*, 1986.

[16] J. Coutaz. *Interfaces homme-ordinateur : conception et réalisation*, pages 139–145. Dunod, 1990.

[17] J. Coutaz. *Interfaces homme-ordinateur : conception et réalisation*, pages 161–186. Dunod, 1990.

[18] I. Crnkovic. *Building Reliable Component-Based Software Systems.* Artech House, Inc., Norwood, MA, USA, 2002.

[19] I. Crnkovic. Building reliable component-based software systems. pages 189–190, 2002.

[20] 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.

[21] J. DeAntoni and J.-P. Babau. A MDA-based approach for real time embedded systems simulation. In *DS-RT*, pages 257–264. IEEE Computer Society, 2005.

[22] J. DeAntoni and J.-P. 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, 2005. http ://www.cs.york.ac.uk/ftpdir/reports/YCS-2005-397.pdf.

[23] J. DeAntoni and J.-P. Babau. Cibermouse design : a case study for SAIA model reuse. *Proceedings of cibermouse team reports, a satellite event of the 27th IEEE International Real-Time Systems Symposium*, pages 8–11, december 2006. http ://www.deantoni.fr.st/deantoni_babau_cibermouse.pdf.

[24] J. DeAntoni and J.-P. Babau. Model driven engineering method for SAIA architecture design. *Ingénierie Dirigée par les modèles (IDM'06)*, 2006.

[25] J. Delacroix. Stabilité et régisseur d'ordonnancement en temps réel. In *Technique et Sciences Informatiques*, volume 13(2), pages 223–250, 1994.

[26] F. DeRemer and H. 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.

[27] M. S. Dias and M. 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.

[28] E. W. Dijkstra. The structure of the THE-multiprogramming system. *Commun. ACM*, 11(5) :341–346, 1968.

[29] 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.

[30] J.-P. B. E. Alemu, D. Bekele. Mda approach for the development of embeddable applications on communicating devices. In *2nd International Workshop on Model-Driven Enterprise Information Systems*, pages 88–97, 2006.

[31] 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.

[32] J.-P. Fassino, J.-B. Stefani, J. L. Lawall, and G. Muller. Think : A software framework for component-based operating system kernels. In *USENIX Annual Technical Conference, General Track*, pages 73–86, 2002.

[33] S. Faucou, A.-M. Déplanche, and Y. Trinquet. An ADL centric approach for the formal design of real time systems. *In Architecture description language, IFIP*, pages 67–82, 2004.

[34] J. 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.

[35] R. T. Fielding. *Architectural styles and the design of network-based software architectures.* PhD thesis, 2000. Chair-Richard N. Taylor.

[36] W. B. Frakes and K. Kang. Software reuse research : Status and future. *IEEE Trans. Softw. Eng.*, 31(7) :529–536, 2005.

[37] R. B. France, D.-K. Kim, S. Ghosh, and E. Song. A uml-based pattern specification technique. *IEEE Transactions on Software Engineering*, 30(3) :193–206, 2004.

[38] 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.

[39] D. Garlan. Software architecture : a roadmap. In A. Finkelstein, editor, *The Future of Software Engineering.* ACM Press, 2000.

[40] D. Garlan. Formal modeling and analysis of software architecture : Components, connectors, and events. In *SFM*, pages 1–24, 2003.

[41] D. Garlan, R. Allen, and J. 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.

[42] D. Garlan and D. Shaw. IEEE recommanded practice for architectural description of software-intensive systems. *ANSI/IEEE Standard 1471*, 2001.

[43] D. Garlan and M. Shaw. An introduction to software architecture. Technical report, Pittsburgh, PA, USA, 1994.

[44] T. Genssler. Pecos in a nutshell, 2002.

[45] C. Ghezzi, M. Jazayeri, and D. Mandrioli. *Fundamentals of software engineering.* Prentice-Hall, Inc., Upper Saddle River, NJ, USA, 1991.

[46] 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.

[47] T. Henzinger, B. Horowitz, and C. Kirsch. Giotto : a time-triggered language for embedded programming. *Proceedings of the IEEE*, 91(1) :84–99, 2003.

[48] D. Isovic and C. Norstrm. Components in real-time systems, 2002.

[49] J.-P. B. J.-C. Tournier, V. Olive. The qinna experiment, a component-based qds architecture for real-time systems. In *Workshop on Architectures for Cooperative Embedded Real-Time Systems, in conjunction with the 25th IEEE RTSS*, 2004.

[50] J.-P. B. J.-C. Tournier, V. Olive. Qinna, a component-based qos architecture. In *8th SIGSOFT symposium on CBSE*, pages 107–122, 2005.

[51] J.-P. B. J.-C. Tournier, V. Olive. A qinna evaluation, a component-based qds architecture for handheld systems. In *ACM Symposium on Applied Computing, SAC 05*, pages 998–1002, 2005.

[52] J. Jezequel, O. Defour, and N. Plouzeau. An MDA Approach to Tame Component Based Software Development. *Formal Methods for Components and Objects, Second International Symposium, FMCO 2003, Leiden, The Netherlands, November 4-7, 2003, Revised Lectures*, 3188.

[53] R. Kazman, G. Abowd, L. Bass, and P. Clements. Scenario-based analysis of software architecture. *IEEE Software*, 13(6) :47–55, November 1996.

[54] R. Kazman, L. J. Bass, M. Webb, and G. D. Abowd. SAAM : A method for analyzing the properties of software architectures. In *International Conference on Software Engineering*, pages 81–90, 1994.

[55] D.-K. Kim, R. France, S. Ghosh, and E. 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.

[56] M. Klein. *A Practitioner's Handbook for Real-Time Analysis : Guide to Rate Monotonic Analysis for Real-time Series*. Kluwer Academic Publishers, 1993.

[57] M. Klein and R. Kazman. Attribute based architectural styles. Technical report, 1999.

[58] 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.

[59] G. E. Krasner and S. 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.

[60] A. Lanusse, S. Gérard, and F. Terrier. Real-time modeling with uml : The accord approach. volume 1618, pages 319–335. Springer, 1999.

[61] K. G. Larsen, P. Pettersson, and W. Yi. UPPAAL in a Nutshell. *Int. Journal on Software Tools for Technology Transfer*, 1(1–2) :134–152, Oct. 1997.

[62] G. B. M. Spuri. Scheduling aperiodic tasks in dynamic priority systems. In *Real-Time Systems*, volume 10, pages 179–210, 1996.

[63] G. Madl, S. Abdelwahed, and D. 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.

[64] N. Medvidovic, P. Oreizy, J. E. Robbins, and R. 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.

[65] N. Medvidovic and R. N. Taylor. A classification and comparison framework for software architecture description languages. *Software Engineering*, 26(1) :70–93, 2000.

[66] Microsoft. Review of windows 3.1 architecture. http ://www.microsoft.com/technet/archive/wfw/1_ch2.mspx, 2005.

[67] Microsoft. .NET. http ://www.microsoft.com/net, 2006.

[68] S. microsystems. JSR 153 : Enterprise JavaBeans 2.1. Technical Report, Java Community Process, 2003.

[69] R. Monroe, A. Kompanek, R. Melton, and D. Garlan. Architectural styles, design patterns, and objects. *Software, IEEE*, 14(1) :43–52, 1997.

[70] OMG-MDA. Model driven architecture guide v1.0.1. *http ://www.omg.org/mda*, 2003.

[71] OMG-OCL. UML 2.0 OCL specification. *http ://www.omg.org/cgi-bin/doc ?ptc/2005-06-06*, 2005.

[72] OMG-UML. Final ftf report mof 2.0 core and uml 2.0 infrastructure finalization task force. *http ://www.omg.org*, 2004.

[73] T. Open Service Gateway initiative. OSGI release platform 4. Technical Report, 2006.

[74] P.H.Feiler, B. Lewis, and S. 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.

[75] project EAST-EEA. Definition of language for automotive embedded electronic architecture. *Version 1.02*, 2004.

[76] M. Shaw, R. DeLine, D. V. Klein, T. L. Ross, D. M. Young, and G. Zelesnik. Abstractions for software architecture and tools to support them. *Software Engineering*, 21(4) :314–335, 1995.

[77] M. Shaw and D. Garlan. *Software Architecture : Perspectives on an Emerging Discipline*. Prentice-Hall, 1996.

[78] J. 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.

[79] D. B. Stewart, R. A. Volpe, and P. K. Khosla. Design of dynamically reconfigurable real-time software using port-based objects. *Software Engineering*, 23(12) :759–776, 1997.

[80] C. Szyperski. *Component Software : Beyond Object-Oriented Programming*. Addison-Wesley Professional, December 1997.

[81] B. Tekinerdogan. Asaam : Aspectual software architecture analysis method. *wicsa*, 00 :5, 2004.

[82] THINK-Team. A think component library. In *think.objectweb.org*, 2007.

[83] R. van Ommering, F. van der Linden, J. Kramer, and J. Magee. The koala component model for consumer electronics software. *Computer*, 33(3) :78–85, 2000.

[84] S. Vestal. Metah user's manual - version 1.27, 1998.

[85] A. Wall. A formal approach to analysis of software architectures for real-time systems. Technical report, September 2000.

[86] K. Wallnau, J. Stafford, S. Hissam, and M. Klein. On the relationship of software architecture to software component technology. *Proc. 6th Workshop on Component-Oriented Programming*, 2001.

[87] K. 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.

[88] A. Wils, J. Gorinsek, S. V. Baelen, Y. Berbers, and K. D. Vlaminck. Flexible component contracts for local resource awareness. In *ECOOP Workshop on resource aware computing*, 2003.

[89] C. working group. Component model specification, v2. In *forge.gridforum.org/sf/projects/cddlm-wg, doc12270, created by S. Shaefer*, 2004.

[90] S. Yovine. Kronos : A verification tool for real-time systems. (kronos user's manual release 2.2).

[91] H. ZIMMERMANN. OS1 Reference Model-The IS0 Model of Architecture for Open Systems Interconnection. *IEEE TRANSACTIONS. ON COMMUNICATIONS*, 28(4) :425, 1980.