Les langages de description d'architecture pour le temps réel

Anne-Marie Déplanche, Sébastien Faucou
Institut de Recherche en Communications et Cybernétique de Nantes (UMR n° 6597)
CNRS / École Centrale de Nantes, École des Mines de Nantes, Université de Nantes
Prenom.Nom@irccyn.ec-nantes.fr

Résumé

En accompagnement de la notion d'architecture logicielle en tant que « perspective haut niveau d'un système logiciel » et reconnue comme le cœur d'une discipline à part entière, des formalismes sont apparus au cours des années 90 : les ADLs (« Architecture Description Languages », soit langages de description d'architecture). Dotés d'une syntaxe et d'une sémantique bien définies, ils ont été proposés en remplacement des nombreuses notations informelles utilisées jusqu'alors pour décrire la structure des systèmes logiciels. De nombreux ADLs ayant été proposés par le milieu académique, l'article présente les concepts et abstractions caractéristiques que la plupart d'entre eux partagent. Il montre ensuite comment, dans le domaine des systèmes temps réel, une démarche architecturale et l'utilisation de langages adaptés tels que les ADLs, qui favorisent une approche globale, sont particulièrement intéressants. Cette argumentation s'appuie sur des ADLs spécialement conçus pour répondre aux besoins spécifiques du domaine temps réel.

1. Introduction

C'est dans les années 70, lorsque les systèmes logiciels ont commencé à franchir un certain seuil de complexité que l'idée de conception architecturale (ou encore « programming-in-the-large »), au sens d'une activité de conception séparée de la conception détaillée (ou encore « programming-in-the-small »), a émergé [8]. La notion d'architecture logicielle, en tant que « perspective haut niveau d'un système logiciel », n'apparaît réellement qu'à partir des années 90 et est alors présentée comme le cœur d'une discipline à part entière [22].

Même si tous s'accordent sur le fait qu'une architecture (logicielle) relève de la structure gros grain d'un système et de l'organisation de logiciels, il n'y a pas de consensus général sur une définition précise de ce terme. Ce terme désigne pour certains la description des structures et organisations des logiciels d'un système (ou d'une classe de système) particulier, tandis que pour d'autres, il désigne une discipline, un champ d'étude. À titre d'exemples, voici quelques définitions rencontrées dans la littérature concernée :

- *« ... it not only reflects how the functional requirements are met, but addresses non-functional requirements, design rationale, architecture style. »* [34]
- *« ... a view of a system that includes the system's major components, the behavior of those components as visible to the rest of the system, and the ways in which the components interact and coordinate to achieve the system's mission. »* [7]
- *« ... the structure or structures of the system, which compromise software components, the external visible properties of those components, and the relationships among them. »* [4]
- *« ... the fundamental organization of a system embodied in its components, their relationships to each other and to the environment, and the principles guiding its design and evolution. »* [25]
- *« While there are numerous definitions of software architecture, at the core of all 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. »* [20]

Enfin, dans [35], Perry et Wolf souhaitent mettre en place les fondations de la recherche future sur les architectures logicielles et proposent pour cela un modèle (accompagné d'une caractérisation des architectures et styles architecturaux logiciels) qu'ils expriment sous forme condensée par le triplet : *Software architecture* = { *Elements, Form, Rationale* } (soit encore, une architecture logicielle est un ensemble d'éléments architecturaux accompagnés de propriétés et relations qui en contraignent le choix et l'organisation, sans oublier les motivations expliquées de l'architecte), plaçant ainsi en quelque sorte l'architecture

logicielle à mi-chemin entre les exigences et la conception.

Les enjeux classiquement mis en avant pour motiver l'émergence de l'approche architecturale sont :
- l'architecture constitue un cadre pour maîtriser la complexité d'un système, dans le but de répondre à toutes les exigences du cahier des charges, qu'elles soient fonctionnelles ou non ;
- l'architecture est une base pour la conception comme pour l'estimation des coûts et la gestion de projet ;
- l'architecture offre un support pour la prise en considération de l'évolutivité et de la réutilisation des systèmes logiciels ;
- l'architecture est une base de raisonnement à des fins d'analyse et de validation.

Ces arguments font apparaître clairement le rôle joué par la description de l'architecture au sein du processus de développement d'un système logiciel : au-delà du résultat de l'étape de conception générale, elle a vocation à servir de référentiel aux étapes suivantes, qui viennent y puiser des informations (extraction d'un modèle formel à des fins d'analyse, extraction de spécifications détaillées à des fins de codage, etc.) et l'enrichir de nouvelles informations en retour. Ainsi, en fonction du stade de développement du système, la description de l'architecture peut être une spécification abstraite du système qu'on cherche à concevoir ou bien une description détaillée du système opérationnel.

Pour permettre la mise en œuvre de cette approche, différentes notations ont été proposées. Trois générations sont identifiées dans la littérature :
- les *langages d'interconnexion de modules* ou MIL (« Module Interconnection Language ») tout d'abord : ils permettent de décrire (au travers des services fournis et requis) les modules logiciels composant une application implémentée, les dépendances entre (les services de) ces modules comme leur implantation physique sur les machines. Ils sont très souvent associés à des bus logiciels assurant l'installation des modules, leur interconnexion comme la gestion de leurs échanges. On trouve dans [36] un panorama de ces langages ;
- les *langages de configuration* ensuite : les entités logicielles ici manipulées sont plus « abstraites » ; on parle généralement de (types de) composants présentant une interface bien définie composée de points d'interaction (au sens large), instanciables, pouvant être primitifs ou composites selon une structuration hiérarchique. La composition est un mécanisme fondamental introduit par ces langages ; elle repose sur l'interconnexion de composants via leurs interfaces. Restant toutefois fortement lié à l'implémentation, le langage de configuration est généralement associé à un langage de programmation détaillée pour le codage des composants primitifs. L'objectif majeur alors visé est celui de la configurabilité, c'est-à-dire la capacité de spécifier et de modifier la configuration d'un système (par ajout, suppression ou remplacement de composants) de manière statique (hors ligne) ou dynamique (en ligne). Dans ce dernier cas, le langage de configuration permet de décrire les changements à opérer et les conditions sous lesquelles ils peuvent avoir lieu. Parmi les langages de configuration les plus connus (en leur temps), on peut citer Conic [26], à l'origine de Darwin [28, 29], ou encore Durra [3] ;
- petit-à-petit, les langages de configuration ont pris une orientation plus conceptuelle et plus formelle, tant pour les composants que pour les connexions entre composants, ceci afin d'accroître les possibilités de raisonnement (évaluation et/ou analyse) au niveau architectural. Une sémantique formelle s'est substituée aux simples conventions de représentation. Est alors apparue, au cours des années 90, la dénomination de *langage de description d'architecture* ou ADL (« Architecture Description Language »). De nombreux ADLs ont été développés par le milieu académique à cette époque : Wright, Darwin, Unicon, Rapide, Aesop, C2 SADL, MetaH, etc. [32]. Il s'est agi là de contributions bien souvent complémentaires mais isolées, rendant difficiles la mise en commun de descriptions architecturales comme l'intégration des outils associés. Le langage ACME [21] est une tentative de réponse, mais dans la pratique il n'offre un support satisfaisant que pour l'échange des aspects structurels des descriptions. On peut cependant penser que l'essor actuel des technologies de transformation de modèle va permettre d'offrir à court terme des techniques adaptées à la résolution de ce problème.

Le domaine des systèmes temps réel affiche des besoins qui justifient actuellement une réelle réflexion sur l'approche de conception architecturale : organisation complexe (présence de fonctionnalités multiples interdépendantes), architectures matérielles réparties, présence de contraintes non-fonctionnelles qui lient intimement éléments logiciels et matériels, utilisation optimisée des ressources, reconfigurabilité dynamique, prédictibilité et donc nécessité de vérification a priori et au plus vite dans le processus de développement, etc. De plus, la généralisation de l'utilisation des systèmes temps réel embarqués dans des domaines comme l'automobile ou l'avionique (où les produits sont déclinés en gamme et sont construits par assemblage de sous-systèmes fournis par différents équipementiers) fait émerger des exigences nouvelles

de flexibilité, réutilisabilité, portabilité, interopérabilité, etc. Pour être respectées, de telles exigences doivent être prises en considération dès la conception de haut niveau. Des travaux de recherche ont été et sont encore menés autour d'un processus de développement « basé architecture » pour la conception de systèmes temps réel et quelques ADLs dédiés sont apparus comme MetaH [5, 39], CLARA [12, 17, 18] , COTRE [16] ou encore EAST-ADL [9]. Récemment, le SAE[1] a normalisé un ADL « temps réel » destiné en premieur lieu au domaine de l'avionique : AADL[2] [37] (« Architecture Analysis and Design Language »). Enfin, ces thèmes ont été au cœur des travaux de l'Action Spécifique CAT (pour Composants et Architectures Temps réel) dont les travaux sont résumés dans [13].

La présentation qui suit est constituée de deux grandes parties. Pour commencer (section 2), nous mettons en place les concepts fondamentaux qui définissent un ADL généraliste et nous les illustrons par un exemple traité avec le langage Wright. Dans la suite, nous ciblons notre propos sur le domaine du temps réel. En section 3 quelques-uns des aspects fondamentaux des systèmes temps réel sont rappelés, qui ne peuvent être ignorés lors de la conception architecturale et qui doivent donc être supportés par un ADL « temps réel ». La complexité résultant de la prise en considération de ces spécificités amène naturellement à la notion de vue architecturale qui est brièvement discutée. Suivent les présentations par le traitement d'un même exemple de deux ADLs temps réel : CLARA (section 4) et MetaH (section 5). En complément, un rapide aperçu de la proposition COTRE (section 6), avant les conclusions (section 7).

2. Les langages de description d'architecture

2.1. Les concepts à la base des ADLs

Les ADLs constituent une classe de langages offrant des abstractions pour la description « gros grain » des systèmes logiciels. Comme indiqué en introduction, de multiples langages appartiennent à cette catégorie, langages qui, pour certains, diffèrent de manière majeure ne serait-ce que par leur syntaxe, leur sémantique, leur expressivité et les buts qu'ils visent. En proposer une définition unique, précise et consensuelle est alors un problème délicat. Pour y répondre, dans un article de 2000 qui fait référence dans le domaine [32], Medvidovic et Taylor ont préféré établir un cadre de classification et de comparaison des ADLs. Dans ce contexte, ils ont proposé un « test » permettant de savoir si un langage est un ADL, fondé sur la présence dans le langage de quatre concepts : **composant**, **interface**, **connecteur** et **configuration**. Nous résumons ci-après les définitions qu'ils donnent de ces concepts.

Un **composant** est une unité de traitement ou de stockage de donnée. C'est une entité dynamique dont l'état évolue au cours du temps, en fonction des sollicitations qu'elle reçoit. Avec les connecteurs, les composants constituent les briques de base utilisées pour décrire le système. Ainsi, pour faciliter la réutilisation d'un composant au sein d'une même architecture ou d'architectures différentes, les ADLs font généralement la différence entre type et instance de composant. Par ailleurs, l'accent est porté sur les possibilités d'utilisation du composants par des tiers, et non sur son fonctionnement interne. Cela explique l'importance donnée à la notion d'interface. Enfin, même si les ADLs ont comme but premier de décrire la structure d'un système, une sémantique est souvent associée au composant, qui décrit son comportement (i.e. les traitements réalisés et événements produits en réaction aux sollicitations reçues à l'interface). Lorsqu'il s'agit d'un composant atomique, le comportement est spécifié en langage naturel, algorithmique ou formel (à des fins de vérification). Pour les composants non atomiques, il est défini par une architecture (configuration) encapsulée.

L'**interface** (ou les interfaces) regroupe l'ensemble des points d'interaction (au sens large) du composant avec son environnement. Pour refléter la nature de l'interaction, ces points sont « orientés » : port en entrée ou en sortie, service offert ou requis, etc. Par ailleurs, un ensemble de contraintes et de propriétés sont associées au composant, qui définissent ses conditions d'utilisation (notion de contrat).

Un **connecteur** est assimilable à un composant spécialisé dans les interactions : il permet de mettre en relation (via leurs interfaces) un ensemble de composants et spécifie les règles qui régissent cette interaction. Ici aussi on distingue type et instance de connecteur. Un connecteur possède une interface constituée de points d'interaction appelés rôles (en référence au rôle joué par le composant qui y est attaché dans l'interaction). Ils sont destinés à être connectés aux points d'interaction des composants (sous réserve de compatibilité entre le point d'interaction et le rôle). La liaison entre les différents rôles est assurée par la "glu" : il s'agit d'une spécification du comportement du connecteur, similaire à la description sémantique d'un composant. Le fait de considérer les connecteurs comme des entités de première classe est une avancée issue du monde des ADLs. On retrouve aujourd'hui cette notion de connecteur dans UML2.

Une **configuration** est un graphe bipartite (instances de) composants - (instances de) connecteurs qui spécifie tout ou partie de l'architecture du système selon un certain point de vue. On peut ainsi avoir une

[1] SAE : Society of Automotive Engineers, `http://www.sae.org`.

[2] Ce langage (promu en partie par l'industrie) faisant l'objet d'un autre chapitre de ce recueil, il n'est pas détaillé ici.

configuration qui exprime la connectivité entre les différentes briques logicielles, une configuration qui exprime les flots de contrôle concurrents qui traversent ces briques et leur répartition sur les ressources d'exécution, etc. La construction d'une configuration est régie par des règles qui garantissent que le système décrit vérifie un certain nombre de bonnes propriétés : typage au niveau des flots de données, absence de deadlocks, respect d'une heuristique de conception (on parle alors de *style* d'architecture, par exemple le style « pipe and filter »). Certaines de ces propriétés sont inhérentes au langage (typage des flots de données), d'autres doivent être exprimées sous forme de contraintes spécifiées au niveau de la configuration (adhésion à un style) et / ou vérifiées a posteriori (absence de deadlock). Les configurations expriment la composition des briques logicielles, celle-ci pouvant être horizontale (interconnexion) ou verticale (abstraction / encapsulation).

2.2. Un exemple

En prenant comme support l'ADL Wright [2], nous allons illustrer les notions introduites ci-dessus.

Wright est un ADL défini par R. Allen et D. Garlan, en 1997, à l'Université de Carnegie-Mellon [2]. Il est destiné à des applications logicielles « classiques » et est donc présenté comme un ADL généraliste. Il offre uniquement une syntaxe textuelle et fournit une base formelle pour la description de configurations architecturales. Son utilisation ici se justifie principalement par la simplicité de ses concepts structurants. Ses caractéristiques majeures peuvent se résumer ainsi :

- les connecteurs font l'objet de spécifications explicites et indépendantes ;
- le comportement abstrait des composants et connecteurs peut être décrit à l'aide d'un sous-ensemble de l'algèbre de processus CSP[3] [24] (nous supposons, pour la suite, que le lecteur possède une connaissance minimum de CSP) ;
- de par la sémantique formelle de cet ADL, la spécification d'une architecture peut donner lieu à un certain nombre de vérifications statiques. Ainsi, une description Wright peut être traduite en CSP puis analysée avec des outils comme FDR [19] pour vérifier la compatibilité d'une connexion (respect d'un même protocole par un composant et un connecteur auquel il est lié) ou encore l'absence de deadlock.

Nous traitons ci-après en Wright l'exemple bien connu du dîner des philosophes. Notre présentation s'inspire de celle donnée dans [30]. La simplicité du problème ne permet pas de présenter l'intégralité des possibilités du langage, le lecteur intéressé se reportera à [2] pour plus de détails et compléments.

Le système à décrire est le suivant : des philosophes se réunissent pour philosopher et dîner. Le dîner est constitué d'un plat de spaghetti qui, selon les coutumes de ces philosophes, se mange à l'aide de deux fourchettes pour un convive. Une table est dressée, qui comporte une assiette par philosophe et une fourchette par assiette. Chaque philosophe pense puis mange puis pense, etc. Il saisit les fourchettes une par une ; il doit posséder les deux fourchettes qui entourent son assiette pour pouvoir manger. Le problème est habituellement de définir un protocole de synchronisation des actions des philosophes de façon à éviter aux philosophes la famine. Étant donné l'objectif de notre exposé, on se concentre ici sur la description de la « structure » du système et la description présentée comporte un deadlock.

Les composants sont les philosophes et les fourchettes. Les connecteurs sont les mains des philosophes destinées à associer mangeurs et outils. Chaque philosophe possède deux bras : son interface est donc composée de deux ports `gauche` et `droite`. Chaque fourchette possède un manche : son interface est donc composée d'un port `manche`. Une main relie un mangeur à un outil : son interface fait état de deux rôles `mangeur` et `outil`. Lorsqu'on met en place une table de philosophes, il s'agit de définir le schéma d'interaction entre les philosophes et les fourchettes, soit encore d'associer chaque bras de chaque philosophe à une fourchette par l'intermédiaire d'une main. Pour ce faire, chaque port d'un philosophe est connecté à une main avec un rôle `mangeur`, tandis qu'une fourchette est connectée à la même main avec un rôle `outil`.

Interface Type $Bras = \overline{prendre} \rightarrow \overline{deposer} \rightarrow Bras \sqcap \S$
Component *Philo*
  Port $Gauche = Bras$
  Port $Droit = Bras$
  Computation $= penser \rightarrow \overline{Gauche.prendre} \rightarrow$
    $\overline{Droit.prendre} \rightarrow manger \rightarrow \overline{Gauche.deposer} \rightarrow$
    $\overline{Droit.deposer} \rightarrow$ **Computation** $\sqcap \S$

Fig. **1. Le composant** *Philo*

Le code Wright qui décrit le composant *Philo* est donné fig. 1. Ce composant a deux ports similaires (les « bras » du philosophe), on définit donc un **Interface Type** pour capturer le comportement commun (sous la forme d'un processus CSP). Outre la définition de son interface, la spécification du comportement du composant est donnée à travers la clause **Computation**.

La description du composant *Fourchette* est donnée par la fig. 2 selon le même principe.

Le code Wright du connecteur *Main* est donné fig. 3. Ce connecteur possède deux rôles dont on spé-

[3]Par rapport à CSP « classique », Wright fait la différence entre les communications initiées par un processus (surlignées) et celles initiées par son environnement (non surlignées). Par ailleurs, § est utilisé comme abréviation de $\checkmark \rightarrow Stop$ pour marquer la terminaison correcte d'un processus.

Component *Fourchette*
Port $Manche = pris \rightarrow depose \rightarrow Manche \sqcap \S$
Computation $= Manche.pris \rightarrow Manche.depose \rightarrow$
Computation $\sqcap \S$

Fig. **2. Le composant** *Fourchette*

Connector *Main*
Role $Mangeur = \overline{prendre} \rightarrow \overline{deposer} \rightarrow Mangeur \sqcap \S$
Role $Outil = pris \rightarrow depose \rightarrow Outil \sqcap \S$
Glue $= Mangeur.prendre \rightarrow \overline{Outil.pris} \rightarrow$ **Glue**
$\sqcap\ Mangeur.deposer \rightarrow \overline{Outil.depose} \rightarrow$ **Glue**
$\sqcap \S$

Fig. **3. Le connecteur** *Main*

cifie le comportement sous la forme de deux processus CSP. Le lien entre ces deux processus est assuré par le processus spécifié dans la clause **Glue** : lorsque le rôle *Mangeur* reçoit le message *prendre*, le message *pris* est émis vers le rôle *Outil*.

Configuration *Diner*
Component *Philo* ...
Component *Fourchette* ...
Connector *Main* ...
Instances
$p1, p2, p3$: *Philo*
$f1, f2, f3$: *Fourchette*
$m11, m12, m21, m22, m31, m32$: *Main*
Attachements
$p1.Gauche$ **as** $m11.Mangeur$
$f1.Manche$ **as** $m11.Outil$
$p1.Droit$ **as** $m12.Mangeur$
$f2.Manche$ **as** $m12.Outil$
$p2.Gauche$ **as** $m21.Mangeur$
$f2.Manche$ **as** $m21.Outil$
$p2.Droit$ **as** $m22.Mangeur$
...
End Diner

Fig. **4. Un extrait de configuration (3 philosophes)**

Le système est décrit au sein de la configuration *Diner* (fig. 4). Après la définition des types de composant et de connecteur (telle que détaillée ci-dessus et non rappelée ici), la rubrique **Instances** permet de spécifier les instances utilisées pour décrire le système. Ces instances sont ensuite connectées dans la rubrique **Attachements** (un port d'une instance de composant à un rôle d'une instance de connecteur).

L'exemple ci-dessus ne nous permet pas d'illustrer toute les possibilités offertes par Wright comme la composition hiérarchique (la clause **Computation** d'un composant est alors définie par une configuration) ou encore la possibilité de définir formellement des styles d'architecture. Dans Wright, un style regroupe un vocabulaire (une ontologie) et un ensemble de contraintes syntaxiques (pour contraindre la topologie d'une architecture relevant de ce style) et sémantique (pour contraindre le comportement abstrait des composants ou connecteurs appartenant à ce style), les contraintes étant exprimées à l'aide de formules de la logique des prédicats d'ordre 1 et de la théorie des ensembles appliquée à des ensembles de base se rapportant aux entités constituant une spécification Wright (par exemple : Connector est l'ensemble des connecteurs d'une configuration, $Traces(P)$ est l'ensemble des traces du processus P, etc.).

D'autre part, dans sa version originale, Wright ne supportait pas la spécification des aspects dynamiques d'une architecture (instanciation /suppression de composant par exemple). Il a été étendu depuis [1], comme *C2* [33] ou encore Darwin [28] qui apportent des réponses plus ou moins complètes à de tels besoins. Pour cela, des événements particuliers « de contrôle » sont introduits et peuvent être référencés dans la description d'un port, permettant ainsi de décrire dans quelles conditions un composant accepte et traite les reconfigurations. Ces événements de contrôle sont utilisés par ailleurs dans une vue séparée, un programme de configuration ou configuror, décrivant comment ces événements déclenchent les reconfigurations. Là encore, en raison de la sémantique formelle associée, des analyses de cohérence peuvent être engagées.

3. ADLs et systèmes temps réel

Comme indiqué en introduction, les systèmes temps réel modernes affichent une complexité croissante du point de vue fonctionnel mais aussi extrafonctionnel (architecture matérielle, support d'exécution, exigences temporelles, de sûreté de fonctionnement, de consommation, interdépendance de ces différentes composantes, etc.). Ceci justifie d'autant l'importance de la conception architecturale pour leur développement et de langages adaptés tels que les ADLs qui favorisent une approche globale et abstraite indispensable. Toutefois, de par leurs spécificités, les concepts de base des ADLs doivent être spécialisés et/ou enrichis afin d'offrir la possibilité d'inclure des informations pertinentes et indispensables au processus de développement consécutif. Nous nous proposons ci-après de mettre en avant quelques-uns des aspects qui nous paraissent fondamentaux. Par les exemples d'ADLs plus particulièrement dédiés au temps réel qui font suite, certains d'entre eux sont illustrés.

Les systèmes temps réel présentent de manière générale un **caractère réactif** vis-à-vis d'événements internes (signal de synchronisation produit par un composant du système, terminaison de l'exécution d'un composant, occurrence d'une situation d'exception, etc.), ou issus de l'environnement contrôlé, ou encore liés au temps (signal d'horloge périodique, expiration d'un chien de garde, etc.). Si l'on souhaite être à même de réaliser des analyses comportemen-

tales, il est primordial de pouvoir décrire, dès le niveau « architecture fonctionnelle », les liens entre les flots de contrôle qui parcourent le système et ces différents événements.

La description de l'**environnement physique** contrôlé ou du moins de ses interactions visibles avec le système relève également des informations à inclure dans la description architecturale d'un système temps réel. D'une part dans le but de pouvoir valider l'architecture proposée par rapport à son futur environnement opérationnel. D'autre part, l'interface de l'environnement n'est pas toujours fixée préalablement à la conception ; il existe alors différentes possibilités d'instrumentation du procédé dont le choix influence la structure et le comportement du système.

Le **temps** est une dimension fondamentale de ces systèmes. Sa manipulation est indispensable dès le niveau architectural, soit qu'il participe au flot de contrôle, soit qu'il quantifie des propriétés temporelles de composants ou de configurations, soit des contraintes temporelles. Issues du cahier des charges, elles sont progressivement dérivées et affinées au cours du développement du système. Les identifier au niveau de la description architecturale assure la documentation et la traçabilité entre les différentes étapes de développement, mais surtout guide les décisions de mise en œuvre (partitionnement, placement, ordonnancement, etc.).

L'**architecture matérielle** comme le **support d'exécution** occupent une place importante dans la conception d'un système temps réel. Leur spécification ne peut donc pas être exclue de la description architecturale et du processus de développement consécutif. D'une part, dans le domaine des applications visées, la définition comme le dimensionnement de ces éléments ne sont pas « libres » (contraintes technologiques diverses, minimisation des coûts, interopérabilité avec des systèmes existants, ressources disponibles limitées, etc.) et constituent déjà en soi un réel problème dès l'étape de conception architecturale. D'autre part, un couplage très fort existe entre logiciels applicatifs et plate-forme support (au sens large). Le comportement du système temps réel, tant qualitativement que quantitativement, dépend bien sûr des services des exécutifs, des middlewares, des protocoles de communication, etc. utilisés, mais aussi de sa nature distribuée ou non, de la capacité de traitement des unités de calcul, de la bande passante des canaux de transmission, des modes de gestion de la mémoire, etc. À moins de procéder par approximations (grossières et parfois insensées), il est donc impossible d'envisager une analyse de performances temporelles comme de sûreté de fonctionnement sur l'architecture du système en construction sans traiter conjointement aspects logiciels et matériels (y compris l'environnement). Sans aller jusqu'à une description aussi riche et fine que celle faite par ailleurs dans le domaine de la conception de circuits (serait-elle exploitable au niveau de la conception architecturale telle que vue ici ?), il apparaît indispensable d'inclure explicitement cette composante dans la description architecturale d'un système temps réel.

Il est fréquent pour un système (de contrôle) temps réel d'adopter des configurations de fonctionnement différentes pendant sa vie opérationnelle. Ces configurations peuvent être liées à des phases différentes de son exploitation ; un système avionique par exemple doit offrir des fonctionnalités spécifiques selon que l'avion est en phase de décollage, de vol ou d'atterrissage. Il peut également s'agir d'évolutions liées à la détection et au traitement de situations d'exception liées à l'occurrence de fautes logicielles ou matérielles du système de contrôle, voire de défaillances d'une partie du procédé ; il faut dans ce cas assurer un service dégradé garantissant l'intégrité du contrôle. Ces différentes configurations sont regroupées dans des **modes de fonctionnement**. Très souvent, le service rendu par un système temps réel se décline ainsi selon différents modes de fonctionnement à activer ou désactiver selon les besoins. La spécification de tels modes et des stratégies appropriées de commutation relève (en partie) du niveau architectural. Il est donc souhaitable qu'un ADL pour le temps réel offre nativement des abstractions adaptées à ces besoins.

Enfin, est-il nécessaire de rappeler que la conception des systèmes temps réel exige non seulement des langages présentant une expressivité adéquate mais aussi des **outils** associés ? Ainsi la possibilité de mener des vérifications de propriétés comportementales (temporelles comme de sûreté de fonctionnement au sens large) au niveau architectural apparaît particulièrement intéressante puisqu'elle profite d'une vision globale du système et doit donc permettre de détecter des erreurs de conception très tôt dans le processus de développement, minimisant ainsi le coût de leur correction. Par ailleurs, la description architecturale d'un système donnée par un ADL est le point d'entrée de l'étape de déploiement. Selon son niveau d'abstraction et d'indépendance vis-à-vis de la plateforme cible, le déploiement d'une description architecturale peut aller de la simple traduction à un travail complexe de construction. L'architecte doit alors être assisté par des outils d'exploration de l'espace des implémentations et de génération/configuration de code. Il apparaît finalement que l'utilisation d'un ADL permet à la fois de dériver d'une même racine commune différents modèles tant pour la vérification/validation que pour l'implantation, et peut donc être un moyen pour de la cassure sémantique ou pour le moins de faciliter la traçabilité entre ces niveaux.

Nous venons d'énumérer un certain nombre d'aspects des systèmes temps réel dont l'expression au niveau architectural est nécessaire. Il est bien entendu qu'une description regroupant simultanément

l'ensemble de ces informations est difficilement réalisable et compréhensible. Il est alors convenu de décomposer la description selon différentes *vues architecturales*, c'est-à-dire selon différentes perspectives pour lesquelles on élimine les entités non concernées au profit de celles directement impliquées dans le point de vue considéré. Le but recherché est de faciliter la démarche de conception en traitant séparément les différents problémes sous-jacents. S'il n'existe pas de réel standard en matière de vue architecturale, le modèle le plus largement accepté par la communauté du génie logiciel (car étroitement lié à UML) est le modèle *« 4+1 views »* (avec ses variantes) [27] qui distingue 5 vues : *Logical view*, *Implementation view*, *Process view*, *Deployment view* et *Use-case view*.

Dans sa thèse [41], A. Wall souligne qu'il est difficile, voire impossible, de décomposer toutes les caractéristiques d'un système en vues parfaitement distinctes. Il préfère donc décliner pour ces différentes vues, trois *« aspects »* particulièrement sensibles pour le domaine des systèmes temps réel (un même aspect étant éventuellement représenté dans plusieurs vues) : *« aspect temporel »*, *« aspect communication »* et *« aspect synchronisation »*.

Dans le même ordre d'idée, dans [17], S. Faucou reprend les trois composantes majeures impliquées dans la conception architecturale d'un système temps réel, à savoir l'*architecture logicielle*, l'*architecture support* et l'*architecture opérationnelle*, et recense pour chacune d'entre elles un certain nombre de vues pertinentes :

- pour l'architecture logicielle : la vue *« composant »* s'attache à la définition individuelle des composants de l'application, tant au niveau de leurs interfaces que de leurs comportements internes ; la vue *« structurelle »* montre l'organisation générale du système, par l'interconnexion des composants et des connecteurs qui expriment la nature des interactions ; la vue *« réactive »* regroupe la description des réactions face aux occurrences d'événements visibles au niveau architectural, établissant ainsi les flots de contrôle des traitements réalisés par le système ;
- pour l'architecture support : la vue *« support logiciel d'exécution »* décrit les services système offerts aux logiciels applicatifs comme les services des exécutifs, d'un middleware, les services de niveau application des protocoles de communication, etc. ; la vue *« support physique d'exécution »* définit et caractérise la constitution physique du système en termes de ressources matérielles et de topologie ;
- pour l'architecture opérationnelle : la vue *« implémentation »* spécifie la projection de l'architecture logicielle sur le support logiciel d'exécution, c'est-à-dire l'implantation des objets et mécanismes de haut niveau de la première sur les objets et services concrets proposés par le second ; la vue *« système »* définit la projection des éléments de la vue d'implémentation sur ceux de la vue support physique d'exécution en termes de placement des éléments logiciels (tâches, variables, messages), d'ordonnancement et optimisation des accès aux ressources matérielles (processeurs et réseaux), de configuration du support logiciel d'exécution, etc. C'est uniquement à partir d'une telle vue (qui est la première à intégrer tous les aspects logiciels, matériels et environnementaux) que l'étude du comportement temporel du système peut être réellement envisagée.

Un point de vue similaire a été adopté par le projet EAST-EEA[4] dont le contexte est celui des systèmes embarqués dans les véhicules terrestres.L'un des objectifs de ce projet a été de définir un langage de description des architectures, EAST-ADL, pour pouvoir décrire de manière plus efficace, et couplée au processus de développement, l'informatique embarquée dans les véhicules. Cinq niveaux d'abstraction ont été proposés et permettent de décrire les fonctionnalités du véhicule et leur organisation depuis un point de vue élevé (ce que voit l'utilisateur : les prestations du véhicule) jusqu'à un niveau très bas (celui des tâches et des messages). On y trouve différentes vues énumérées ci-après (les quatre premières relèvent du logiciel d'application tandis que les trois dernières sont dédiées à l'implémentation) : *« vehicule view »* , *« functional analysis architecture »* , *« functional design architecture »* , *« logical architecture »* , *« hardware architecture »* , *« technical architecture »* et *« operational architecture »* .

Dans les deux sections qui suivent, nous présentons deux ADLs temps réel : CLARA et MetaH. Ces présentations reposent sur un exemple (fictif et simple) d'application de pilotage numérique décrit dans le paragraphe suivant. Pour chaque langage, nous évaluons la façon dont il répond à la liste d'exigence établie cidessus.

Le système à spécifier possède deux entrées (une mesure réalisée sur le procédé piloté et une consigne transmise par l'opérateur fixant l'état à atteindre) et une sortie (une action à réaliser sur le procédé piloté). C'est un système échantillonné. Fonctionnellement, le système de pilotage doit : traduire le format brut délivré par le capteur en un format adapter aux calculs les calculs, puis calculer la commande (fonction de la mesure, de la valeur courante de la consigne et de l'état estimé du procédé), estimer le nouvel état et enfin traduire la commande en un ordre compréhensible par l'actionneur. La période d'échantillon-

[4]EAST-EEA : Electronic Architecture and Software Technology - Embedded Electronic Architecture. Projet ITEA de juin 2001 à juin 2004, impliquant un consortium de constructeurs automobiles, d'équipementiers automobiles et d'universitaires : `http://www.east-eea.net`.

nage est fixée à 500ms avec une gigue acceptable de 20ms. On souhaite par ailleurs que l'action soit notifiée à l'actionneur au plus tard 250ms après l'instant d'échantillonnage.

La figure 5 est la description d'une architecture logicielle pour ce système, décrite à l'aide la syntaxe graphique de CLARA. Cette syntaxe intuitive nous permet de présenter notre architecture candidate avant d'entrer dans les détails des langages. Nous avons choisi de découper le système en quatre composants, correspondant aux quatre activités suivantes : traduction de la mesure, calcul de la commande, calcul du nouvel état et traduction de la commande. Le système est périodique et les synchronisations entre les composants sont ensuite établies par le flot de données. Ce flot n'établit pas d'ordre pour l'exécution des deux dernières activités (calcul du nouvel état et traduction de la commande), ce choix étant laissé aux étapes ultérieures de la conception (choix du nombre des unités d'ordonnancement, de la politique sélectionnée et des valeurs des paramètres associés).

4. CLARA

La spécification complète de l'exemple traité est donnée en annexe A.1.

4.1. Présentation

CLARA [12, 18] (Configuration LAnguage for Real-time Application) est un ADL créé dans les années 90 au sein de l'équipe « Systèmes Temps Réel » de l'IRCCyN. Comme le montrent les figures 5 et 8 qui décrivent (quasiment) la même chose, il possède une syntaxe graphique et une syntaxe textuelle. Il permet la description à un haut niveau d'abstraction de l'architecture fonctionnelle d'applications temps réel. L'accent est particulièrement porté sur la spécification des synchronisations induites par les flots de contrôle et de données qui traversent les activités constitutives de l'application.

Les composants principaux sont les **activités**, qui peuvent être atomiques (associées à un traitement séquentiel fini) ou composées (englobent une configuration). Chaque activité possède deux interfaces : une interface implicite de contrôle constituée d'un port `START` et d'un port `END`, utilisée pour synchroniser le lancement (resp. se synchroniser sur la terminaison) du traitement associé à l'activité ; une interface explicite de transfert, composée de ports (orientés), utilisée pour permettre aux flots de données et de contrôle de traverser l'activité. Cette interface est spécifiée lors de la définition du type d'activité (fig. 6), de même que le comportement interne (plus d'explication sur ce sujet sont données en 4.2).

Au rang des autres composants figurent les variables et ressources partagées, ainsi que les **générateurs d'occurrences**. Ces derniers sont utilisés

```
TYPE_ACTIVITY T_CalculCommande;
    VAR_IN etat, consigne : t_etat;
           mesure : t_cmd_in;
    VAR_OUT commande : t_cmd_out;
    BEHAVIOR receive(mesure) -> receive(etat)
        -> receive(consigne)
        -> call(calculCmd, 10ms, 20ms)
        -> send(commande);
END_ACTIVITY;
```

FIG. **6. Définition d'un type d'activité**

comme source des flots de contrôle et de données. Ils produisent des occurrences selon une certaine loi et peuvent être internes au système (cas d'une horloge par exemple) ou externes (cas d'un périphérique attaché à un canal d'interruption par exemple). La figure 7 montre la définition d'un type de générateur périodique (utilisé ici pour cadencer le système).

```
TYPE_GENERATOR T_Horloge_500ms;
    PERIODIC 500ms;
    SIGNAL_OUT out;
    BEHAVIOR CLOCK;
END_GENERATOR;
```

FIG. **7. Définition d'un type de générateur d'occurrences**

Au sein des configurations, les (instances des) activités sont reliées par des **liens** construits autour de **protocoles**. Un protocole correspond à la notion de « glue » vue précédemment : il spécifie la politique de synchronisation entre l'entité jouant le rôle de producteur et l'entité jouant le rôle de consommateur dans cette interaction. Les **connecteurs** permettent de connecter un port d'une activité à un protocole (s'il s'agit d'un port en sortie, l'activité jouera le rôle producteur, s'il s'agit d'un port en entrée, l'activité jouera le rôle consommateur). Un connecteur peut être simple (association 1-1) ou complexe (association n-m, de diffusion ou de sélection). Dans une configuration, les liens sont spécifiés dans la rubrique `LINKS`. Ainsi, fig. 8, on a spécifié que :

- de `capture.mesure_out` vers `calculCommande.mesure_in` est établi un lien de transfert de données du type rendez-vous ;
- une donnée publiée par `calculCommande.commande` est : transmise vers `action.commande` via un rendez-vous ; transmise vers `majEtat.commande` via une boîte aux lettres ;
- de `H1.out` vers `capture.START` est établi un lien de transfert de signaux du type mémorisé (i.e. production asynchrone, consommation synchrone) ;
- etc.

La figure 8 correspond à la spécification du composant **système**, qui constitue la racine d'une des-

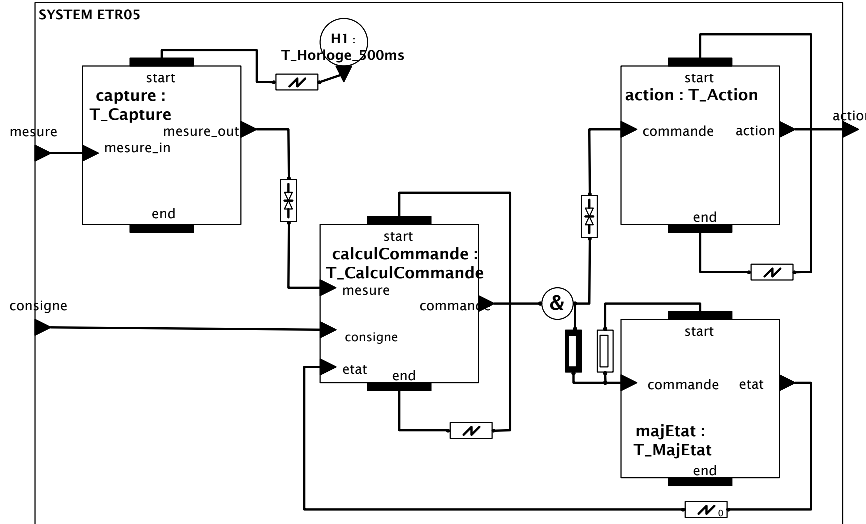

FIG. **5. Description de l'architecture fonctionnelle de l'exemple en CLARA (syntaxe graphique)**

```
SYSTEM Etr05;
VAR_IN mesure : t_mesure; consigne : t_etat;
VAR_OUT action : t_action;
GENERATOR H1 : T_Horloge_500ms;
ACTIVITY
    capture : T_Capture;
    calculCommande : T_CalculCommande;
    majEtat : T_MajEtat;
    action : T_Action;
LINKS
    // liens d'interface
    mesure TO capture.mesure_in;
    consigne TO calculCommande.consigne;
    action.action TO action;
    // liens inter-activites
    capture.mesure_out TO
        calculCommande.mesure : RDVd;
    calculCommande.commande TO
        (action.commande : RDVd
        & majEtat.commande : BAL);
    majEtat.etat TO calculCommande.etat : RAFO;
    // liens d'activation
    H1.out TO capture.start : MEM;
    calculCommande.end TO calculCommande.start : MEM;
    action.end TO action.start : END;
    majEtat.commande TO majEtat.start : INC;
CONSTRAINTS
    Abs(capture.mesure_in.cnf) < 25ms;
    480ms < capture.mesure_in.cnf < 520ms;
    0ms < capture.mesure_in.cnf :
          action.action.cnf     < 250ms;
END_SYSTEM.
```

FIG. **8. Définition de la configuration**

cription CLARA. Comme tout composant, il possède une interface. Son comportement, il est donné sous la forme d'une configuration : instanciation de composants encapsulés (rubriques `GENERATOR` et `ACTIVITY`), spécification des liens entre eux (rubrique `LINKS`) et finalement expression de contraintes (rubrique `CONSTRAINTS`).

4.2. Aspects réactifs

Dans CLARA, les aspects réactifs occupent une place prépondérante. Les flots de contrôle et de données du système sont ainsi entièrement modélisé. On spécifie comment ils passent d'un composant à un autre à travers les liens, mais également comment ils traversent les composants : lors de la définition d'un type d'activité atomique (cf. fig. 6), dans la rubrique `BEHAVIOR`, on spécifie le comportement sous la forme d'une séquence alternant interaction avec les flots de contrôle et de donnée (primitives `send`, `receive`, etc.) et appel de fonctions de l'application (primitive `call`). Implicitement, chaque exécution d'une telle séquence est précédée d'une synchronisation sur le port `START` et suivie d'une synchronisation sur le port `END`. Comme évoqué précédemment, les flots ont pour source des générateurs d'occurrences qui permettent de modéliser des sources d'interruptions matérielles (horloge ou périphérique). Une description CLARA explicite donc complètement le lien entre l'écoulement du temps, les événements de l'environnement perçus par le système de pilotage, et les fonctions logicielles qu'il exécute en réaction.

4.3. Environnement physique

CLARA ne permet pas de modéliser l'environnement physique. L'utilisation de générateurs d'occurrences permet cependant de modéliser les sources de signaux ou de données externes au système. Le choix des protocoles qui contrôlent les liens entre les générateurs externes et l'interface du système permet de modéliser différentes politiques d'interaction : interruption, scrutation, etc.

4.4. Temps

Dans une description CLARA, le temps apparaît à 3 niveaux. Premièrement, au niveau des générateurs

d'occurrences, pour spécifier les instants de production. Ensuite, dans la description du comportement d'une activité, l'architecte donne une estimation du meilleur et du pire temps d'exécution des fonctions utilisateur appelées (par exemple `calculCmd` figure 6). Elles deviennent des exigences à respecter pour la phase d'implémentation. Enfin, le temps intervient au niveau des contraintes portant sur une configuration (section `CONSTRAINTS` de la figure 8). Ces contraintes, relatives ou absolues, portent sur la durée séparant deux occurrences. Là encore, il s'agit d'exigence qui doivent guider les étapes ultérieures du développement. Concernant l'exemple, on a exprimé (cf. fig. 8) que la lecture d'une mesure par l'instance d'activité `capture` doit être suivie par l'émission d'une action par l'instance `action` dans un intervalle de 250ms (contrainte relative). On a également exprimé une contrainte absolue : la première lecture du capteur doit avoir lieu moins de 25ms après le démarrage du système (événement associé implicitement à la date 0). Pour des raisons de lisibilité, l'expression graphique des contraintes n'a pas été portée sur la fig. 5. Dans [18], nous expliquons comment vérifier la cohérence entre les différentes grandeurs relatives au sein d'une description CLARA.

4.5. Support d'exécution

CLARA n'offre pas de moyen de décrire le support d'exécution, que ce soit au niveau matériel (nombre et caractéristiques des calculateurs, des mémoires, interconnexion entre ces éléments) ou au niveau logiciel (services fournis par l'exécutif).

4.6. Modes de fonctionnement

Dans [12], E. Durand fait une proposition d'extension de CLARA pour l'expression des modes de fonctionnement. Basiquement, un mode de fonctionnement est un composant hiérarchique qui peut contenir des objects CLARA classiques et éventuellement des sous-modes de fonctionnement. Au sein du composant système, ainsi qu'au sein de chaque mode comportant des sous-modes, un réacteur est décrit, qui spécifie les conditions de commutation entre les sous-modes. Cette proposition n'a pas encore été intégrée au langage.

4.7. Outils

Le langage CLARA a pour vocation de servir de base à différents travaux académiques menés au sein de l'équipe « Systèmes Temps Réel » de l'IRCCyN. Les outils développés pour et autour de lui n'ont donc pas dépassé le stade de prototype et servent principalement de preuve de faisabilité. Ont ainsi été étudiés :

- la compilation vers les réseaux de Petri temporels pour la vérification de propriétés avec hypothèse d'architecture matérielle à ressources infinies [12] ;
- la projection d'une architecture CLARA sur un support d'exécution OSEK/VDX distribué et une technique de validation par simulation d'un modèle SDL de l'architecture opérationnelle résultante [17] ;
- l'utilisation des outils ROMÉO, TINA et CADP pour la vérification de propriétés de sûreté et la vérification de la cohérence des grandeurs temporelles (budget d'exécution vs. contraintes) [18] ;
- l'utilisation des outils de transformation de modèle ATL pour la projection d'architecture CLARA sur des supports d'exécution temps réel (VxWorks, RTAI) [10].

5. MetaH

La spécification complète de l'exemple traité est donnée en annexe A.2.

5.1. Présentation

MetaH est un langage développé au cours des années 90 par S. Vestal et son équipe chez Honeywell, dans le cadre de l'initiative « Domain Specific Software Architecture » [5]. Leurs travaux constituent une des bases de la norme AADL. Le langage permet de décrire l'architecture logicielle, l'architecture matérielle et la projection de la première sur la seconde *i.e.* l'architecture opérationnelle, à des fins de validation et de génération de code. Plus précisément, les systèmes visés sont : critiques, temps réel, complexes et multiprocesseurs.

```
process ICalculCommande is
    etat: in port ETR05.T_ETAT;
    consigne: in port ETR05.T_ETAT;
    mesure: in port ETR05.T_CMD_IN;
    commande: out port ETR05.T_CMD_OUT;
end ICalculCommande;
process implementation ICalculCommande.Default is
    calculCmd: subprogram;
paths
    <<Normal>> := calculCmd;
attributes
    calculCmd'SourceTime := 20ms;
    self'ComputePath := Normal;
end ICalculCommande.Default;
```

Fig. **9. Définition d'un type d'interface et d'implémentation de process**

Les composants principaux sont les **processes**. Un process est une unité d'ordonnancement (et optionnellement un espace d'adressage protégé). Il s'agit donc bien d'une entité de l'architecture opérationnelle, contrairement aux activités atomiques CLARA qui peuvent être projetées selon différentes stratégies (regroupement, éclatement, etc.). On commence par définir un type d'interface (cf. fig. 9), qui décrit les

ports d'entrée et de sortie du process (les variables importées et exportées), les événements qu'il peut produire et les **package** et **monitor** qu'il rend disponible (ce sont les mêmes notions que dans Ada, langage auquel MetaH est fortement lié). On peut ensuite définir différents types d'implémentations correspondant au type d'interface. Dans une implémentation, on définit les flots de contrôle du process (rubrique `paths`) ainsi que différents attributs relatifs par exemple au temps d'exécution. Dans la définition de l'implémentation d'un process, aucune information n'est obligatoire. Cela correspond à la philosophie des ADLs : une description architecturale accompagne le développement du système et se complète donc au fur et à mesure.

Lors de l'instanciation d'un process (cf. fig. 10), des informations supplémentaires peuvent être précisées. Il faut ainsi spécialiser le process en `aperiodic` ou `periodic` et le cas échéant préciser sa période. Un process apériodique est activé sur occurrence d'un événement matériel ou logiciel, tandis qu'un process périodique est activé par la couche exécutif MetaH (MetaH est un ADL « implementation dependent », qui vise une plate-forme d'exécution particulière. A contrario, CLARA et WRIGHT sont « implementation independant »). Le comportement d'un process est défini par un ensemble de `path` qui sont des séquences finies sans point de blocage (voir paragraphe 5.2).

```
macro Etr05 is
    mesure: in port ETR05.T_MESURE;
    consigne: in port ETR05.T_ETAT;
    action: out port ETR05.T_ACTION;
end Etr05;
macro implementation Etr05.Default is
    capture: periodic process ICapture.Default;
    calculCommande: periodic process
      ICalculCommande.Default; ...
connections
    capture.mesure_in <- mesure; ...
    action <<- action.action;
    calculCommande.mesure <<- capture.mesure_out; ...
    calculCommande.etat <- majEtat.etat;
attributes
    ...
    majEtat'Period := 500ms;
    action'Period := 500ms;
    action'Deadline := 250ms;
end Etr05.Default;
```

FIG. **10. Description d'une macro**

Le composant **macro** (cf. fig. 10) permet de structurer une spécification d'architecture logicielle. Il encapsule une configuration pour en faciliter la manipulation. Il possède une clause `connections` pour spécifier les interconnexions entre composants. En ce qui concerne la signalisation, il est possible de connecter un événement à un process apériodique (déclenchement) ou à un mode (commication). En ce qui concerne les flots de données, MetaH supporte la connexion « undelayed » notée `<<-` où la donnée est rendue disponible au consommateur dès la terminaison du producteur, et la connexion « delayed » notée `<-` où la donnée est rendue disponible au consommateur uniquement à la fin de la période du producteur.

Le composant **mode** correspond à la notion de mode de fonctionnement. Un mode encapsule une configuration de composants dont l'activation est liée à celle du mode. Les commutations de mode se font sur occurrence d'événement (logiciel ou matériel).

```
application Etr05 is
    macro Etr05.Default on
      processor MPC565.Default;
connections
    Etr05.mesure <- MPC565.port1;
    Etr05.consigne <- MPC565.port2;
    MPC565.port3 <<- Etr05.action;
attributes
    MPC565'ClockPeriodMax := 64us;
end Etr05;
```

FIG. **11. Description de l'application.**

Enfin, le composant **application** (cf. fig. 11) est la racine d'un nœud du système. Il permet en de mettre en correspondance les éléments de l'architecture logicielle et ceux de l'architecture matérielle de ce nœud : un événement logiciel peut être associé à une interruption, un port logiciel peut être associé à un port matériel, etc.

5.2. Aspects réactifs

Dans MetaH, les flots de contrôle sont associés aux processu. Ils sont activés soit périodiquement, soit en réaction à un événement. Ils peuvent être suspendus lorsqu'une instance de process accède à un moniteur partagé, et stoppés suite à la terminaison d'un process ou à une commutation de mode. Le comportement d'un process est décrit via la clause `path` comme un ensemble de séquence finie d'appel à des **subprograms** et d'accès à des monitor ou package. Les données en entrée sont lues avant l'activation et les données en sortie sont publiées après la terminaison. Les flot de données interprocess induisent donc de simple contrainte de précédence entre process. Si l'on impose que chaque subprogram implante un algorithme (et donc termine) et si la spécification est complète (i.e. les clauses `paths` décrivent bien tous les chemins possibles), on déduit l'ensemble des flots de contrôle. Par conséquent, les réactions de l'application à un événement, qu'il s'agisse d'un événement produit par l'environnement (et relayé via une interruption) ou l'expiration d'une horloge, sont toutes décrites.

Par ailleurs, les contraintes imposées sur le comportement des process permettent de s'assurer que les modèles utilisés pour les analyses (en particulier d'ordonnançabilité) sont cohérents avec l'implémentation. Ce problème est traité dans [40], en utilisant une approche formelle originale (par analyse de modèles obtenus par instrumentation du code).

5.3. Environnement physique

Comme CLARA, MetaH permet d'expliciter uniquement les points de communications entre le système de contrôle et le système contrôlé par le biais des événements matériels, des ports matériels et des périphériques (cf. paragraphe 5.5).

5.4. Temps

Le temps dans MetaH est exprimé via différents attributs : période et échéance d'un process, temps d'exécution d'un path. L'attribut période se passe de commentaire. L'attribut échéance, associé à une instance de process, est le seul moyen d'exprimer des contraintes de temps. Tandis que CLARA vise à décrire l'architecture fonctionnelle des applications et doit donc offrir des mécanismes abstraits d'expression de contraintes destinées à être raffinées, MetaH décrit l'architecture opérationnelle et permet donc uniquement l'expression de contraintes primitives, manipulables directement par l'ordonnanceur (avec une politique EDF ou DM) et par les outils d'analyse d'ordonnançabilité de l'atelier construit autour du langage. Enfin, les temps d'exécution (associés aux `path` ou aux `subprogram`) sont donnés sous la forme de constantes correspondant aux pires temps d'exécution. Ils sont également destinés à être utilisé par les outils d'analyse d'ordonnançabilité.

5.5. Support d'exécution

MetaH est l'un des rares ADL qui permet la description de l'architecture matérielle des systèmes. Sans entrer dans les détails, les principaux composants disponibles pour ce faire sont : **processor** (une carte pour laquelle on spécifie le modèle de processeur, l'exécutif et la chaîne de développement associée, ainsi qu'une interface permettant la liaison entre l'architecture matérielle et l'architecture logicielle), **device** (un périphérique), **memory** (bloc de mémoire, éventuellement partagée, caractérisé par sa taille en octet et le schéma d'adressage utilisé), **channel** (bus point-à-point bidirectionnel utilisé pour interconnecter différents processor et / ou différents devices) et enfin **system** (racine de la spécification). Ces composants possèdent bien entendu des interfaces, constituées de port et d'événements, destinés à être finalement mises en correspondance avec leurs homologues logicielles. La description du support d'exécution est très peu utilisée par les outils d'analyse et permet avant tout la construction et le déploiement automatisés d'images binaires correspondant aux spécifications.

5.6. Modes de fonctionnement

Les composants logiciels mode « définissent des configurations en-ligne alternatives ». Lorsqu'un mode devient actif, la configuration qu'il englobe est mise en place (activation de process, mise à jour des tables de la couche exécutif MetaH). Inversement, lorsqu'un mode devient inactif, la configuration en cours est « détruite » (arrêt des process concernés). Des modes frères dans l'arborescence associée à l'architecture logicielle sont mutuellement exclusifs mais l'utilisation de macro permet d'avoir plusieurs modes actifs au même instant. Enfin, comme évoqué ci-avant, les commutations de mode sont provoquées par l'occurrence d'événements.

5.7. Outils

Autour de MetaH, un atelier complet existe, qui permet la validation de l'architecture proposée et la génération automatique de l'application associée. La figure 12 issue de [39] en présente la structure. Elle illustre bien l'intérêt d'un développement orienté architecture, qui permet de produire automatiquement, à partir d'une même source, les modèles de validation et l'implémentation.

Comme le précisent les auteurs, l'atelier est un prototype, toujours en développement [6]. Il a cependant été utilisé avec succès sur quelques cas d'études industriels (en particulier un système de guidage de missile [31]) et a montré sur ces expériences que l'approche architecturale, lorsqu'elle est supportée par un atelier logiciel adéquate, permet de diminuer les temps et coût de conception et développement d'un système temps réel.

6. COTRE

Destiné à proposer une méthode de conception accompagnée d'outils logiciels (de simulation, analyse d'ordonnançabilité, vérification comportementale, fiabilité, évaluation de performances, etc.) pour les systèmes avioniques, le projet COTRE[5] [16] a choisi de s'appuyer sur un ADL pour décrire les architectures de ces systèmes.

De façon à permettre le développement simultané du niveau utilisateur et du niveau vérification (plus formel), deux langages frères ont été définis : UCOTRE (pour « User COTRE ») proche de l'architecte et VCOTRE (pour « Verification COTRE ») proche des formalismes de vérification. Le but initial était de fusionner les deux branches au sein d'un unique langage. Finalement, COTRE est présenté aujourd'hui comme un profil AADL [15]. En tirant profit des mécanismes d'extension offerts par AADL, COTRE offre ainsi des constructions pour la spécification de :

- contrat, c'est-à-dire l'expression de propriétés requises par un composant (suppositions / exigences sur le comportement de l'environnement)

[5]COTRE (« COmposant Temps REel ») est un projet RNTL qui a débuté en janvier 2002 pour une durée de 2 ans et dont le consortium est composé de AIRBUS, TNI-Valiosys, ENSTB et FéRIA.

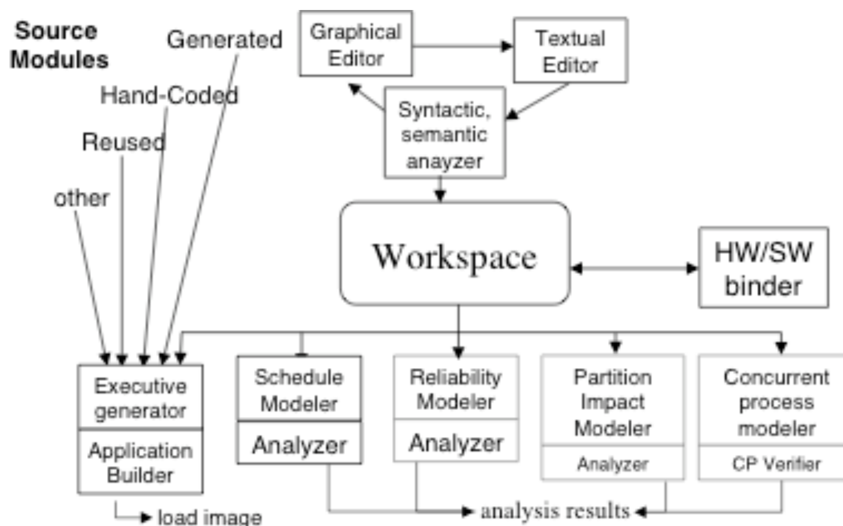

FIG. **12. L'atelier construit autour de MetaH [39]**

et de propriétés garanties par le composant (relatives à son comportement). Ces propriétés s'expriment à l'aide d'assertions de haut niveau ou d'équivalences comportementales ;
– propriétés (attributs) relatives à l'implémentation d'un composant élémentaire : période d'activation, priorité, phase, etc. ;
– comportement d'un composant sous la forme de machines de Mealy.

Les travaux dont nous avons connaissance ont porté sur l'utilisation de UCOTRE comme profil AADL. Actuellement, la cohérence sémantique entre UCOTRE et AADL, d'une part, et UCOTRE et VCOTRE, d'autre part, est à l'étude.

Le langage VCOTRE [14] est quant à lui présenté comme un langage intermédiaire, interface entre l'ADL et les différents formalismes de vérification utilisés dans le projet : systèmes de transitions, automates temporisés et réseaux de Petri. Ainsi, dans VCOTRE, suivant la philosophie des ADLs, l'architecture statique est décrite sous forme d'une structure hiérarchique de composants ; les composants présentent une interface ; leur composition utlise des connecteurs. Les composants élémentaires sont soit des processus applicatifs, soit des composants de communication et de synchronisation (sémaphore, boîte aux lettres, etc.) issus de la spécification ARINC 653. Le comportement d'un processus est décrit par un système de transitions étiquetés. L'interface est constituée d'un ensemble de ports de communication (typés) et de l'ensemble des propriétés qui forment le contrat du composant. Ces propriétés peuvent être choisies parmi un ensemble de propriétés prédéfinies ou exprimées directement à l'aide du langage d'assertions supporté par l'outil de vérification visé.

7. Conclusion et perspectives

Avec ce chapitre, notre intention est d'introduire simplement et sans ambition d'exhaustivité les ADLs et plus spécifiquement les ADLs pour le temps réel. Cette présentation doit donc être considérée comme préalable à une étude plus approfondie. Après avoir situé la notion d'architecture et les enjeux qui s'y rapportent, nous avons identifié et illustré les principaux concepts-clés des ADLs généralistes. Les caractéristiques de ces langages pouvant constituer une réponse aux problèmes posés par l'augmentation de la complexité des systèmes temps réel, nous nous sommes ensuite attardé sur les ADLs temps réel. Si les concepts de base et les abstractions correspondantes sont bien sûr à conserver, nous avons souligné que pour différentes raisons, ils devaient être adaptés et enrichis pour satisfaire aux spécificités du domaine visé. C'est ce que nous avons voulu ensuite montrer en présentant ensuite CLARA et MetaH, assortis d'un même exemple illustratif. Par manque de place, le projet COTRE a été abordé de manière succincte. Pour la même raison, nous avons passé sous silence d'autres travaux étiquetés ADL temps réel comme Unicon [42] ou traitant de la conception architecturale des systèmes temps réel comme Basement [23]. Quant à AADL, un autre chapitre lui est consacré (voir le chapitre « Les ADL du point de vue de l'industrie » dans ce volume).

D'une manière générale, il faut admettre que la multiplicité des propositions académiques en matière d'ADLs (et notamment pour le temps réel) n'a pas favorisé l'aboutissement des travaux associés tout comme leur adoption par le milieu industriel. La mise en place du langage AADL en tant que standard SAE devrait maintenant constituer un référence pour le

domaine temps réel. Par les mécanismes d'extension qu'il offre, son adaptation locale est possible. C'est d'ailleurs la voie retenue par le projet COTRE et il nous semble qu'il faille tendre maintenant vers une telle unification.

Nous pensons que l'ADL et la description d'architecture ne doivent pas être cantonnés à la seule étape de conception architecturale. Ainsi, dans le cas de MetaH, la description architecturale (et les attributs qu'elle porte) est le point de départ pour la génération d'une implémentation mais aussi pour la dérivation des divers modèles exploités par les outils d'analyse. Pour réduire davantage la cassure sémantique entre les modèles de V&V et l'implantation, ou du moins pour faciliter la traçabilité entre ces niveaux, il faut penser l'ADL pour qu'il constitue « l'épine dorsale » des différentes étapes du processus de développement. D'une manière schématique, une étape extrait de la description d'architecture alors disponible les informations qui la concernent, puis, en retour, l'enrichit des données qu'elle a produites. Un tel retour d'informations au niveau de la description simplifie l'interprétation des résultats et peut permettre à terme une utilisation transparente des techniques formelles.

Des travaux en matière d'outillage d'ADL doivent bien sûr être poursuivis afin d'assister l'architecte tant sur le plan de la V&V que du déploiement. Ainsi, sur ce dernier point, la tâche est plus ou moins difficile selon le degré de couplage qu'affiche l'ADL avec les aspects opérationnels. Dans le cadre de CLARA, le niveau abstrait choisi pour décrire une architecture fait que les décisions relatives au déploiement restent entièrement à prendre : il s'agit bien alors de « construire » et non de « traduire » une implémentation. Pour un système temps réel, il s'agit schématiquement de définir les tâches, les trames, leur placement sur les ressources de calcul et de communication, la configuration résultante des supports d'exécution, etc. Pour une même architecture, l'espace des implémentations peut donc être important et des outils d'exploration et d'aide à la décision doivent être proposés. Signalons que le couplage outils de déploiement et langages de niveau conception (même s'ils ne s'agit pas à proprement parler des ADLs) est au cœur de la problématique « Model Integrated Computing » [38].

Enfin, dans la mesure où les problèmes ici soulevés sont difficiles à traiter, il apparaît incontournable de les traiter dans un cadre bien délimité et restrictif. L'identification de classes d'applications, de familles d'architectures, de patrons architecturaux, ou encore de styles architecturaux contribue à une plus grande spécialisation des langages et modèles utilisés et favorise la mise en place de schémas de conception éprouvés. À notre connaissance, il s'agit d'un point encore peu abordé dans le domaine des ADLs temps réel.

Références

[1] R. Allen, R. Douence, and D. Garlan. Specifying and analyzing dynamic software architectures. In *Conference on Fundamental Approaches to Software Engineering (Lisbonne, Portugal)*, 1998.

[2] R. J. Allen. *A Formal Approach to Software Architecture*. Thèse de doctorat, School of Computer Science, Carnegie Mellon University, 1997.

[3] M. Barbacci et al. Durra : a structure description language for developing distributed applications. *Software Engineering Journal*, 8(2) :83–94, 1993.

[4] L. Bass, P. Clements, and R. Kazman. *Software architecture in practice*. Addison-Wesley, 1998.

[5] P. Binns, M. Englehart, M. Jackson, and S. Vestal. Domain-Specific Software Architectures for Guidance, Navigation and Control. *International Journal of Software Engineering and Knowledge Engineering*, 6(2), 1996.

[6] P. Binns and S. Vestal. Hierarchical Composition and Abstraction in Architecture Models. In Dissaux et al. [11], pages 35–50.

[7] P. Clements. Attractions in software architecture. Technical Report CMU/SEI-96-TR-008, Software Engineering Institute, 1996.

[8] F. de Remer and H. Kron. Programming-in-the-large versus programming-in-the-small. *IEEE Transactions on Software Engineering*, 2(2), 1976.

[9] V. Debruyne, F. Simonot-Lion, and Y. Trinquet. EAST-ADL : an Architecture Description Language. Validation and Verification aspects. In Dissaux et al. [11], pages 181–196.

[10] J. Delatour, F. Thomas, G. Savaton, and S. Faucou. Modèle de plate-forme pour l'embarqué : première expérimentation sur les noyaux temps réel. In *Ingénierie Dirigée par les Modèles*, 2005.

[11] P. Dissaux, M. Filali Amine, P. Michel, and F. Vernadat, éditeurs. *Architecture Description Languages*, volume 176 of *IFIP*. Springer, 2004.

[12] E. Durand. *Description et vérification d'architecture temps réel : CLARA et les réseaux de Petri temporels*. Thèse de doctorat, École Centrale de Nantes, Nantes, France, 1998.

[13] A. Déplanche et al. Rapport de synthèse – AS 195 - Composants et Architectures Temps réel. Technical Report RI2005_1, IRCCyN, 2005.

[14] P. Farail et al. The COTRE project : how to model and verify real-time architecture? In *2nd European Congress on Embedded Real-Time Software (ERTS'2004)*, Toulouse, 2004.

[15] P. Farail and P. Gaufillet. COTRE as an AADL profile. In Dissaux et al. [11], pages 167–179.

[16] J. Farines et al. The COTRE project : rigirous software development for real-time systems in avionics. In *27th IFAC/IFIP/IEEE Workshop on Real-Time Programming*, mai 2003.

[17] S. Faucou. *Description et construction d'architectures opérationnelles validées temporellement*. Thèse de doctorat, Université de Nantes, Nantes, France, 2002.

[18] S. Faucou, A. Déplanche, and Y. Trinquet. An ADL-centric approach for the formal design of real-time systems. In Dissaux et al. [11], pages 67–82.

[19] Formal Systems (Europe) Ltd. *Failures-Divergence Refinement – FDR2 User Manual*, 2003.

[20] D. Garlan. Software Architecture : a Roadmap. In A. Finkelstein, éditeur, *International Conference on Software Engineering - Proceedings of the conference on the Future of Software Engineering*, pages 91–101. ACM Press, 2000.
[21] D. Garlan, R. Monroe, and D. Wile. ACME : An Architecture Description Interchange Language. In *CASCON'97*, pages 169–183, novembre 1997.
[22] D. Garlan and M. Shaw. *An introduction to software architecture*. World Scientific Publishing, 1993.
[23] H. Hansson, H. Lawson, and M. Strömberg. Basement : a distributed real-time architecture for vehicle applications. *Real-Time Systems*, 11(3) :223–244, 1996.
[24] C. Hoare. *Communicating Sequential Processes*. Prentice Hall International, 1985.
[25] IEEE. IEEE Recommended Practice for Architectural Description of Software-Intensive Systems - ANSI/IEEE Standard 147-2000, 2001.
[26] J. Kramer, J. Magee, and M. Sloman. Constructing distributed systems in Conic. *IEEE Transactions on Software Engineering Software*, 15(6) :663–675, 1989.
[27] P. Kruchten. Architectural blueprints - the "4+1" view model of software architecture. *IEEE Software*, 12(6), 1995.
[28] J. Magee, N. Dulay, S. Eisenbach, and J. Kramer. Specifying Distributed Software Architectures. In *European Software Engineering Conference*, volume 989 of *LNCS*, pages 137–153. Springer, 1995.
[29] J. Magee and J. Kramer. Dynamic structure in software architectures. In *Fourth ACM SIGSOFT Symposium on the Foundations of Software Engineering*, pages 3–14, octobre 1996.
[30] R. Marvie. *Les intergiciels*, chapitre Langages de description d'architectures - Un état de l'art. Hermès, 2005.
[31] D. McConnell, B. Lewis, and L. Gray. Reengineering a single threaded embedded missile application onto a parallel processing platform using metah. *Real-Time Systems*, 14(1) :7–20, 1998.
[32] N. Medvidovic and R. Taylor. A Classification and Comparison Framework for Software Architecture Description Language. *IEEE Transactions on Software Engineering*, 26(1) :70–93, Jan. 2000.
[33] P. Oreizy, N. Medvidovic, and R. N. Talylor. Architecture-Based Runtime Software Evolution. In *International Conference on Software Engineering*, pages 177–186. IEEE Computer Society, 1998.
[34] F. Paulisch. Software architecture and reuse : an inherent conflit ? In 3rd *International Conference on Software Reuse*, 1994.
[35] D. Perry and A. Wolf. Foundations for the study of software architecture. *ACM SIGSOFT Software Engineering Notes*, 17(4) :40–52, 1992.
[36] R. Prieto-Diaz and J. Neighbors. Module interconnection languages. *The Journal of Systems and Software*, 6(4) :307–334, 1986.
[37] Society of Automotive Engineer. *AS5506 - Architecture Analysis and Design Language (AADL)*, 2004.
[38] J. Sztipanovits and G. Karsai. Model-integrated computing. *Computer*, 30(4) :110–111, 1997.
[39] S. Vestal. *MetaH User's Manual - version 1.27*. Honeywell Technology Center, Minneapolis, MN, USA, 1998.
[40] S. Vestal. Modeling and Verification of Real-Time Software Using Extended Linear Hybrid Automata. In *Lfm2000 : Fifht NASA Langley Formal Method Workshop*, pages 83–94, 2000.
[41] A. Wall. *Architectural modeling and analysis of complex real-time systems*. Thèse de doctorat, Department of Computer Science and Engineering, Mälardalen University, 2003.
[42] G. Zelesnik. The UniCon Language Reference Manual. Technical report, School of Computer Science, Carnegie Mellon University, 1996.

A. L'exemple (presque) complet

A.1. En CLARA

```
TYPE_ACTIVITY T_Capture;
    VAR_IN mesure_in : t_mesure;
    VAR_OUT mesure_out : t_cmd_in;
    BEHAVIOR receive(mesure_in)
        -> call(InToOut, 100us, 500us)
        -> send(mesure_out);
END_ACTIVITY.
TYPE_ACTIVITY T_CalculCommande;
    VAR_IN etat, consigne : t_etat;
        mesure : t_cmd_in;
    VAR_OUT commande : t_cmd_out;
    BEHAVIOR receive(mesure) -> receive(etat)
        -> receive(consigne)
        -> call(calculCmd, 10ms, 20ms)
        -> send(commande);
END_ACTIVITY;
TYPE_ACTIVITY T_MajEtat;
    VAR_IN commande : t_cmd_out;
    VAR_OUT etat : t_etat;
    BEHAVIOR receive(commande)
        -> call(calculEtat, 100ms, 250ms)
         -> send(etat);
END_ACTIVITY;
TYPE_ACTIVITY T_Action;
    VAR_IN commande : t_cmd_out;
    VAR_OUT action : t_action;
    BEHAVIOR receive(commande)
        -> call(cmdToAction, 0.5ms 1ms)
        -> send(action);
END_ACTIVITY;
TYPE_GENERATOR T_Horloge_500ms;
    PERIODIC 500ms;
    SIGNAL_OUT out;
    BEHAVIOR CLOCK;
END_GENERATOR;

SYSTEM Etr05;
VAR_IN mesure : t_mesure; consigne : t_etat;
VAR_OUT action : t_action;
GENERATOR H1 : T_Horloge_500ms;
ACTIVITY
    capture : T_Capture;
    calculCommande : T_CalculCommande;
    majEtat : T_MajEtat;
    action : T_Action;
LINKS
    // liens d'interface
    mesure TO capture.mesure_in;
    consigne TO calculCommande.consigne;
    action.action TO action;
    // liens inter-activites
    capture.mesure_out TO
        calculCommande.mesure : RDVd;
    calculCommande.commande TO
        (action.commande : RDVd
        & majEtat.commande : BAL);
    majEtat.etat TO calculCommande.etat : RAFO;
    // liens d'activation
    H1.out TO capture.start : MEM;
```

```
    calculCommande.end TO calculCommande.start : MEM;
    action.end TO action.start : END;
    majEtat.commande TO majEtat.start : INC;
CONSTRAINTS
    Abs(capture.mesure_in.cnf) < 25ms;
    480ms < capture.mesure_in.cnf < 520ms;
    0ms < capture.mesure_in.cnf :
          action.action.cnf     < 250ms;
END_SYSTEM.
```

A.2. En MetaH

```
with type package ETR05;

-- declaration des interfaces des processus
process ICapture is
    mesure_in: in port ETR05.T_MESURE;
    mesure_out: out port ETR05.T_CMD_IN;
end ICapture;
process ICalculCommande is
    etat: in port ETR05.T_ETAT;
    consigne: in port ETR05.T_ETAT;
    mesure: in port ETR05.T_CMD_IN;
    commande: out port ETR05.T_CMD_OUT;
end ICalculCommande;
process IMajEtat is
    commande: in port ETR05.T_CMD_OUT;
    etat: out port ETR05.T_ETAT;
end IMajEtat;
process IAction is
    commande: in port ETR05.T_CMD_OUT;
    action: out port ETR05.T_ACTION;
end IAction;

-- declaration des implementations des processus
process implementation ICapture.Default is
    InToOut: subprogram;
paths
    <<Normal>> := inToOut;
attributes
    inToOut'SourceTime := 500us;
    self'ComputePath := Normal;
end ICapture.Default;
process implementation ICalculCommande.Default is
    calculCmd: subprogram;
paths
    <<Normal>> := calculCmd;
attributes
    calculCmd'SourceTime := 20ms;
    self'ComputePath := Normal;
end ICalculCommande.Default;
process implementation IMajEtat.Default is
    calculEtat: subprogram;
paths
    <<Normal>> := calculEtat;
attributes
    calculEtat'SourceTime := 250ms;
    self'ComputePath := Normal;
end IMajEtat.Default;
process implementation IAction.Default is
    cmdToAction: subprogram;
paths
    <<Normal>> := cmdToAction;
attributes
    cmdToEtat'SourceTime := 1ms;
    self'ComputePath := Normal;
end IAction.Default;

-- declaration de la macro englobant
-- l'architecture logicielle
macro Etr05 is
    mesure: in port ETR05.T_MESURE;
    consigne: in port ETR05.T_ETAT;
    action: out port ETR05.T_ACTION;
end Etr05;
macro implementation Etr05.Default is
    capture: periodic process ICapture.Default;
    calculCommande: periodic process
        ICalculCommande.Default;
    majEtat: periodic process IMajEtat.Default;
    action: periodic process IAction.Default;
connections
    capture.mesure_in <- mesure;
    calculCommande.consigne <- consigne;
    action <<- action.action;
    calculCommande.mesure <<- capture.mesure_out;
    action.commande <<- calculCommande.commande;
    majEtat.commande <<- calculCommande.commande
    calculCommande.etat <- majEtat.etat;
attributes
    capture'Period := 500ms;
    calculCommande'Period := 500ms;
    majEtat'Period := 500ms;
    action'Period := 500ms;
    action'Deadline := 250ms;
end Etr05.Default;

-- liaison avec l'archi matérielle
application Etr05 is
    macro Etr05.Default on processor MPC565.Default;
connections
    Etr05.mesure <- MPC565.port1;
    Etr05.consigne <- MPC565.port2;
    MPC565.port3 <<- Etr05.action;
attributes
    MPC565'ClockPeriodMax := 64us;
end Etr05;
```