HAL
open science

ContrACT : une méthodologie de conception et de développement d'architectures de contrôle de robots

Robin Passama

To cite this version:

Robin Passama. ContrACT : une méthodologie de conception et de développement d'architectures de contrôle de robots. RR-10025, 2010, pp.45. lirmm-00505309

HAL Id: lirmm-00505309

https://hal-lirmm.ccsd.cnrs.fr/lirmm-00505309v1

Submitted on 23 Jul 2010

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.

ContrACT : une méthodologie de conception et de développement d'architectures de contrôle de robots

Auteur : Robin Passama
email : passama@lirmm.fr
Rapport LIRMM n°: RR-10025

1) Introduction et contexte

Le contexte du travail donnant lieu au présent document sont les projets ANR ASSIST et PROSIT. Le projet PROSIT vise à développer un robot télé-échographe support notamment à un ensemble d'études scientifiques quant aux stratégies de commande utilisables (commandes basées sur la téléopération, commandes référencées vision et commandes mixtes incluant vision et téléopération) afin d'évaluer leur pertinence par rapport aux besoins intrinsèques de ses utilisateurs (équipes médicales) ainsi que leur « qualité » au regard de leur robustesse et de leur performance. Le projet ASSIST vise quant à lui à développer un robot d'assistance aux personnes à mobilité réduite, support notamment à un ensemble d'études scientifiques quant aux stratégies de commande utilisables pour le contrôle d'un robot à deux bras manipulateurs, potentiellement en interaction directe avec des humains.

Le développement du logiciel supportant l'exécution des différentes stratégies de commande relatives à différents modes de fonctionnement (autonome, téléopéré, mixte) peut s'avérer particulièrement complexe et couteux en temps si un minimum de rigueur n'est pas adopté par les acteurs du projet. En ce sens, l'architecture de contrôle du robot, qui définit la manière dont le logiciel du robot est structuré est la pierre angulaire autour de laquelle doivent s'articuler les développements logiciels. Au niveau conceptuel, elle est le plan qui permet d'organiser les développements et au niveau concret, elle est le résultat de l'intégration de l'ensemble des briques logicielles développées par les différents acteurs en fonction de ce plan.

Comme toute architecture logicielle, l'architecture de contrôle doit répondre à un ensemble de propriétés « non fonctionnelles » classiques en génie logiciel comme : la définition de briques logicielles les plus indépendantes possibles les unes des autres afin de faciliter leur ré-utilisabilité ; la possibilité de composer ces briques de manière « flexible » afin de favoriser l'évolutivité (capacité à évoluer, à être modifié) du logiciel résultant. La ré-utilisabilité permet également de minimiser le travail des acteurs du projet en réduisant les temps de développement alors que l'évolutivité permet de construire l'architecture de manière incrémentale (du moins complexe au plus complexe) ce qui permet de mieux maitriser son test et sa maintenance et donc au final sa fiabilité. Ces propriétés favorisent également une meilleure organisation du travail des acteurs du projet en permettant de définir clairement leur responsabilité vis à vis du développement logiciel (quelles briques reviennent à chacun) et leurs interactions (comment se passe l'intégration des briques au sein de l'architecture globale). A la différence de la plupart des architectures logicielles classiques, l'architecture de contrôle doit également prendre en compte les aspects temps-réel (contraintes sur les temps et périodes d'exécution des boucles de commande par exemple) et critiques (réaction prioritaires à certains événements survenant durant la mission) intrinsèques aux application robotiques.

Pour fournir une architecture de contrôle qui puisse respecter à la fois les objectifs et contraintes sus-cités, le LIRMM a proposé de réutiliser une partie des travaux antérieurs dans le domaine menés au sein du laboratoire et de les faire évoluer afin de les adapter aux besoins spécifiques des projets PROSIT et ASSIST, notamment pour la prise en compte de la téléopération, de la supervision et de l'utilisation d'algorithmes non bornés temporellement.

2) Élaboration de la méthodologie de conception / développement de l'architecture de contrôle

La première étape consiste à définir un ensemble cohérent d' « artefacts » qui permettent de définir la méthode et les outils utilisés pour concevoir et développer une architecture de contrôle. Nous avons adopté une approche générique pour définir cette méthodologie, c'est-à-dire que cette dernière est indépendante des projets PROSIT et ASSIST, même si bien sur elle prend en comte les besoins inhérents à ceux-ci. Ceci se justifie par deux raisons : la possibilité de réutiliser la méthodologie pour des projets futurs tant au sein du LIRMM qu'au sein des autres équipes partenaires ; la volonté de ne pas limiter nos options afin de pouvoir s'adapter aux modifications susceptibles d'apparaitre dans le projet (notamment l'ajout de nouvelles stratégies de commandes ou des modification majeures dans l'approche adoptée pour la supervision du système, etc.). La proposition résultante, qui regroupe ces « artefacts » sous le terme de méthodologie *ContrACT* (Control Architecture Creation Technology), est décomposé en trois artefacts présentés par la suite:

1. Une architecture de contrôle générique
2. Un middleware robotique
3. Un outil logiciel (IDE) support au développement logiciel

2.1) Architecture de contrôle générique

L'architecture de contrôle générique est vue comme un plan de travail global, indépendant de l'application cible et de ses diverses modification potentielle au cours d'un projet. Elle définit la méthode utilisée pour concevoir le logiciel de contrôle. Il s'agit avant tout d'un support quant à la manière de communiquer les choix de conception réalisé pour un projet donné. En définissant les grand principes de conception à suivre, elle a pour but de favoriser la cohérence des choix de conception qui seront réalisés par les différentes équipes concernées mais aussi de faciliter la compréhension du fonctionnement du logiciel de contrôle.

Nous présentons cette architecture suivant 3 points complémentaires :
1. L'organisation conceptuelle du cycle perception-décision-action
2. Le modèle de programmation des briques logicielles
3. La décomposition logicielle d'une architecture de contrôle

2.1.1) Organisation conceptuelle du cycle perception-décision-action

L'organisation conceptuelle est la brique de base permettant de définir les grandes lignes sur la façon dont sont conçus les architecture de contrôle suivant la méthodologie *ContrACT*. Il s'agit d'une vision relativement simple de l'architecture en terme de placement des traitements mis en jeu dans le cycle Perception-Décision-Action du robot.

La premier élément de cette organisation est la notion de « couche », représentant **un niveau d'abstraction dans la prise de décision**. L'organisation conceptuelle définit deux couches (cf. fig. 1):

- La couche **exécutive**, la plus basse, est charger de réaliser les décisions prises dans la couche supérieure. Elle contient des traitements de calcul (e.g. planification de trajectoire, de mission), des traitement de mémorisation (e.g. accès base de donnée), les boucles de commande, les boucles d'observation d'événements ainsi que les interactions (e.g. via le réseau) avec les systèmes/agents (e.g. robot ou opérateur) externes. Cette couche est alimentée en **perceptions** grâce aux dispositifs physique d'entrée (e.g. réseau, capteurs, périphérique d'entrée utilisateur, caméras, etc.) et génère des **actions** grâce aux dispositifs physique de sortie (e.g. réseau, actionneurs, écrans, périphériques avec retour de force, etc.). Suivant la nature des perceptions et actions (i.e. suivant les dispositifs utilisés) la couche exécutive « travaille » à la fois dans un monde continu discrétisé (e.g. loi de commande) ou par essence discret (interaction réseau). L'ensemble des *blocs* (i.e. traitements, algorithmes, codes) présents dans cette couche sont considérés comme parallèles : ils n'ont pas de relation directe entre eux.

- La couche **décisionnelle**, la plus haute, contient les traitements liés une prise de décision **réactive**, c'est-à-dire qu'en fonction d'un **contexte** (état du robot, de l'environnement et de la mission) qu'elle reconstitue à partir des **événements** provenant de la couche inférieure et elle définit les **réactions** à adopter (sous la forme de requête d'**activation/paramétrage** vers la couche inférieure), qui seront effectivement réalisées au sein de la couche exécutive. Au sein de la couche décisionnelle La couche décisionnelle « travaille » dans un monde exclusivement discret (événementiel). L'ensemble des *blocs* (appelés superviseurs car chargé de la supervision du système) présents dans cette couche suivent une relation hiérarchique représentant la décomposition de la prise de décision au sein du robot depuis le bloc chargé de la supervision générale du système (i.e. le point d'entrée de l'application), appelé *Superviseur global*.

Notons que qu'avec cette organisation un ordre provenant d'un opérateur humain est considéré comme une **perception** (issue d'une interaction particulière, par exemple à travers une IHM) qui sera transformée en un événement relayé vers le niveau décisionnel. Notons également qu'un calcul traditionnellement lié au niveau décisionnel dans les architecture de contrôle, comme notamment la **planification**, est effectué **dans la couche exécutive**. C'est la **décision de planifier** (une mission, un trajectoire, un chemin) qui est effectuée (i.e. lancée et paramétrée) dans la couche décisionnelle en fonction des objectifs que se fixe le robot à un moment donné.

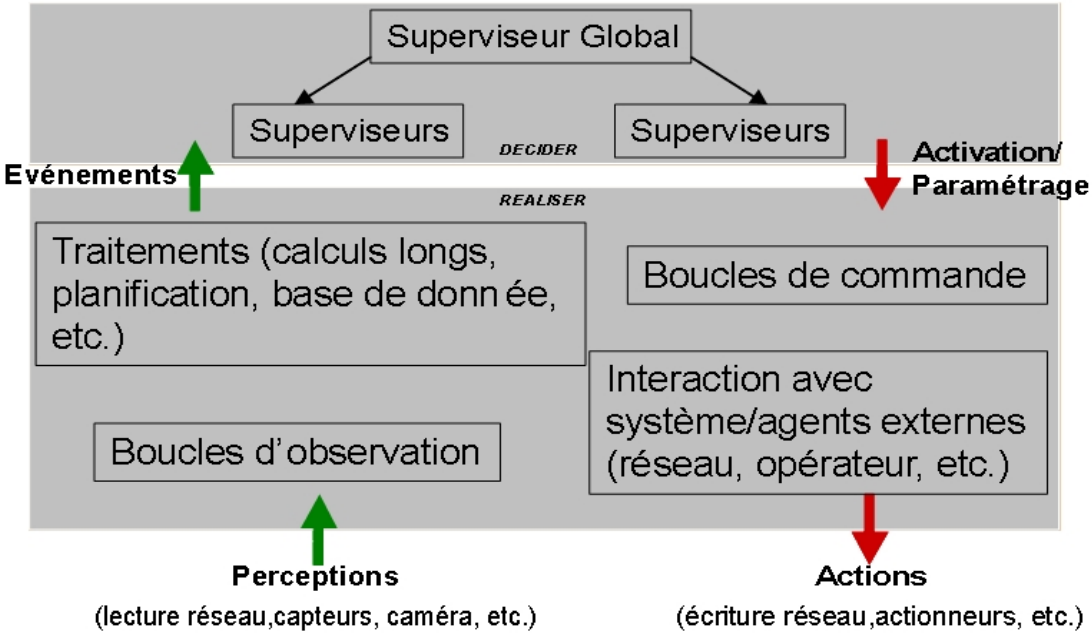

Figure 1 : Organisation conceptuelle du cycle perception-décision-action

2.1.2) Modèle de programmation des briques logicielles

Dès lors que nous considérons l'organisation conceptuelle précédente, il est nécessaire de savoir comment seront définis les *blocs* constitutifs des deux couches. Ces *blocs* seront au final implanté par du code logiciel il convient donc de définir un **modèle de programmation** de ces *blocs*. Ce modèle de programmation définit les **briques logicielles** qui permettent de concrétiser ces *blocs*. Nous avons fait deux choix notables dans la définition de ce modèle de programmation:

1. Nous considérons qu'un *bloc* peut être concrétisé à partir d'une composition d'un nombre indéfini (par défaut) de briques logicielles, voire par une unique brique logicielle.
2. Dans un souci de simplicité et d'homogénéité nous souhaitons avoir un modèle de brique homogène quel que soit le type de *bloc* considéré. C'est le contenu, la structure de leur interface et la manière de composer les briques qui détermineront les *bloc* considérés.

Nous avons intitulé la brique de base de notre modèle de programmation **module**. Un module (cf. fig.2) est une entité (i.e. brique) logicielle qui interagit avec d'autres modules à travers son interface, son comportement interne n'étant pas directement accessible ou modifiable depuis l'extérieur du module.

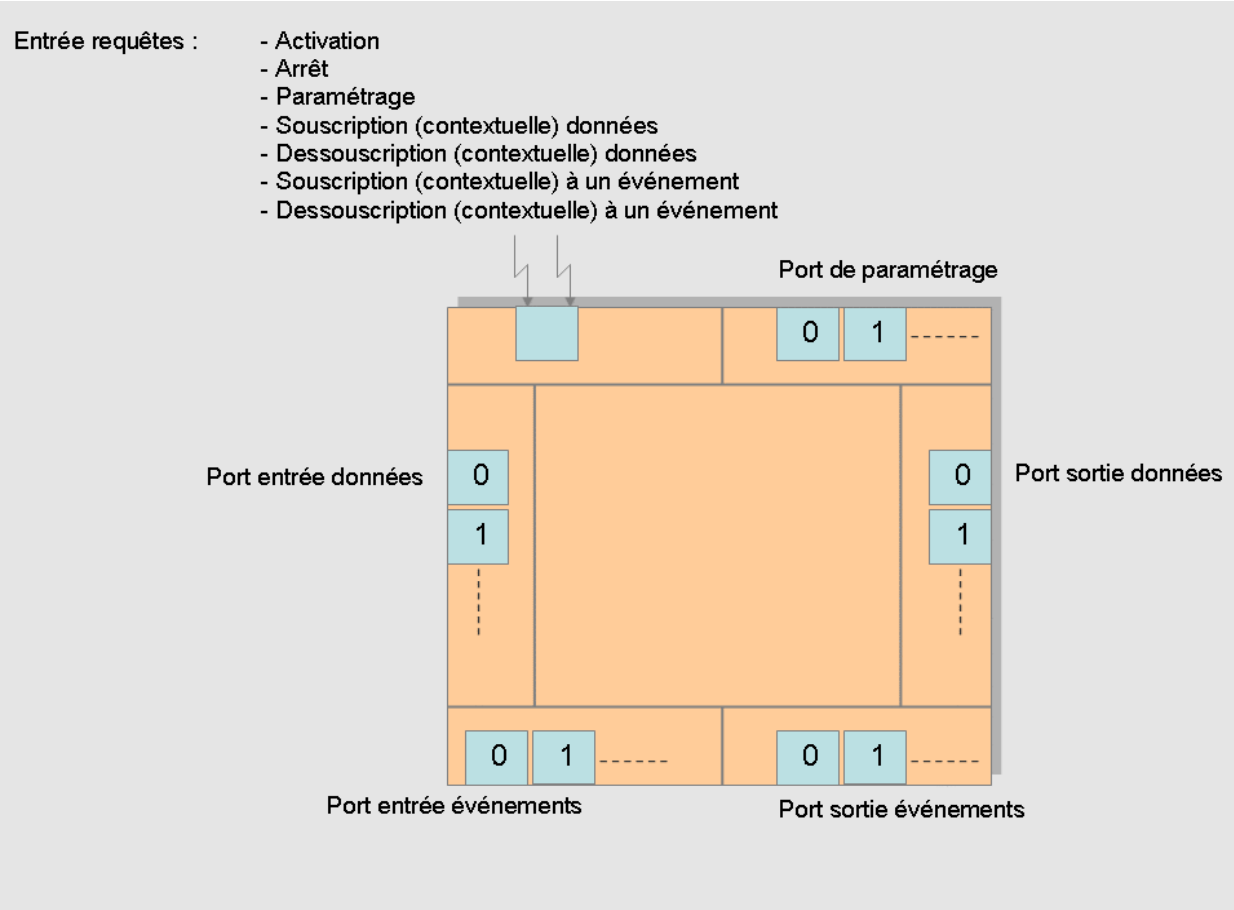

Figure 2 : Modèle de module

Un module possède comme caractéristiques principales (cf. fig.2) :

- Un **port de requête**, c'est-à-dire un point d'entrée de messages qui permettent à d'autres modules de contrôler son activité (démarrage, arrêt) et sa configuration (fixation d'un paramètre, abonnement de ses flux).
- Un ensemble de **ports de paramètres**, c'est-à-dire un ensemble de points d'entrée qui permettent chacun à d'autres modules de fixer la valeur d'un de ses paramètres « publics ».
- Un ensemble de **ports d'entrée de données**, qui correspondent chacun à un flux de données entrant dans le module. Ces ports peuvent être connectés à des ports de sortie de données d'autres modules.
- Un ensemble de **ports de sortie de données**, qui correspondent chacun à un flux de données sortant du (générés par) module. Ces ports peuvent être connectés à des ports d'entrée de données d'autres modules.
- Un ensemble de **ports d'entrée d'événements**, qui correspondent chacun à un flux de d'événements entrant dans le module. Ces ports peuvent être connectés à des ports de sortie de d'événements d'autres modules.
- Un ensemble de **ports de sortie de d'événements**, qui correspondent chacun à un flux de d'événements sortant du (générés par) module. Ces ports peuvent être connectés à des ports d'entrée d'événements d'autres modules.

Excepté le port de requête qui est le point d'entrée global du module tous les autres ports sont associé à une **donnée** qui elle même respecte un **type de donnée**. Les différences entre les types de ports présentés sont les suivantes:

- Le **port de requête**, les **ports de paramètres** et les **ports d'entrée d'événements** sont des ports déclencheurs, ce qui signifie que l'arrivée d'un message va potentiellement engendrer une réaction du module.
- Par opposition, les **ports d'entrée de données** sont non déclencheurs, l'arrivée d'un message (i.e. une donnée) n'engendrera pas de réaction de la part du module. Une fois activé par ailleurs, le module pourra seulement venir lire son port d'entrée de donnée pour savoir s'il contient une donnée.
- Les **ports de sortie de données** (respectivement **d'événement**) ne peuvent générer des données que vers les **ports d'entrées** (respectivement **d'événement**) auxquels ils sont **connectés**. C'est le comportement interne du module qui décide quand produire des données et des événements en mettant à jour les données associés aux ports de sortie correspondant. Le comportement interne du module n'a pas connaissance des modules vers lesquels ces données et événements produits seront acheminés.
- Les ports d'**entrée/sortie** de **données/événements** peuvent être connectés soit par le comportement interne du module qui aurait alors une connaissance particulière de ses relations avec les autres modules, soit plus couramment par un module tier. Ainsi sont créés des flux d'événement et de données au sein du système.
- La connexion et la déconnexion des ports d'**entrée/sortie** de **données/événements** des modules se fait sur demande au module producteur (celui qui produit la donnée ou l'événement) via des **requêtes de configuration**. Ainsi l'ensemble des connexions au sein de l'architecture logicielle sont dynamiques (i.e. susceptibles de changer durant l'exécution).
- Les **connexions entre ports de données** sont **continues** (il faut un désabonnement explicite auprès du module producteur pour que le flux doit interrompu) et **cycliques** (un module reçoit les données produites que toutes les x activations du module producteur, avec x compris entre 1 et l'infini).
- Les **connexions entre ports d'événements** sont soit **continues**, soit **éphémères** (le désabonnement du consommateur se fait dès lors que l'événement a été produit et diffusé).
- Les ports de **sortie** de **données** (respectivement **événements**) peuvent être connectés à un nombre quelconque de ports d'entrée de **données** (respectivement **événements**), mais un port d'entrée de **données** (respectivement **événements**) ne peut être connecté qu'à un seul port de **sortie** de **données** (respectivement **événements**).
- Les **ports de paramètres** servent à recevoir des données correspondant aux nouvelles valeurs des paramètres modifiés. Il n'y a pas de notion de flux et de connexion pour les ports de paramètre, ils sont donc utilisables uniquement dans une relation un-à-un de type appel de procédure entre deux modules.

Ce modèle de programmation permet d'établir à la fois des flux de données et des flux d'événements entre modules, qui sont régulièrement utilisés (rarement de façon conjointe) dans les architectures de contrôle, en plus des relations classique de type activation/arrêt/paramétrage. Grâce au **port de requête** les interconnexions entre modules peuvent être modifiées par un tierce module (cf. figure 3) au cours de leur exécution. Cette **dynamicité des relations** de flux entre modules (illustrée dans la figure 3) constitue la caractéristique essentielle sur laquelle est construite l'ensemble de l'architecture de contrôle logicielle. Une architecture bâtie à partir de ce modèle de

programmation résulte in fine à une interconnexion dynamique de modules échangeant données et événements via leurs ports.

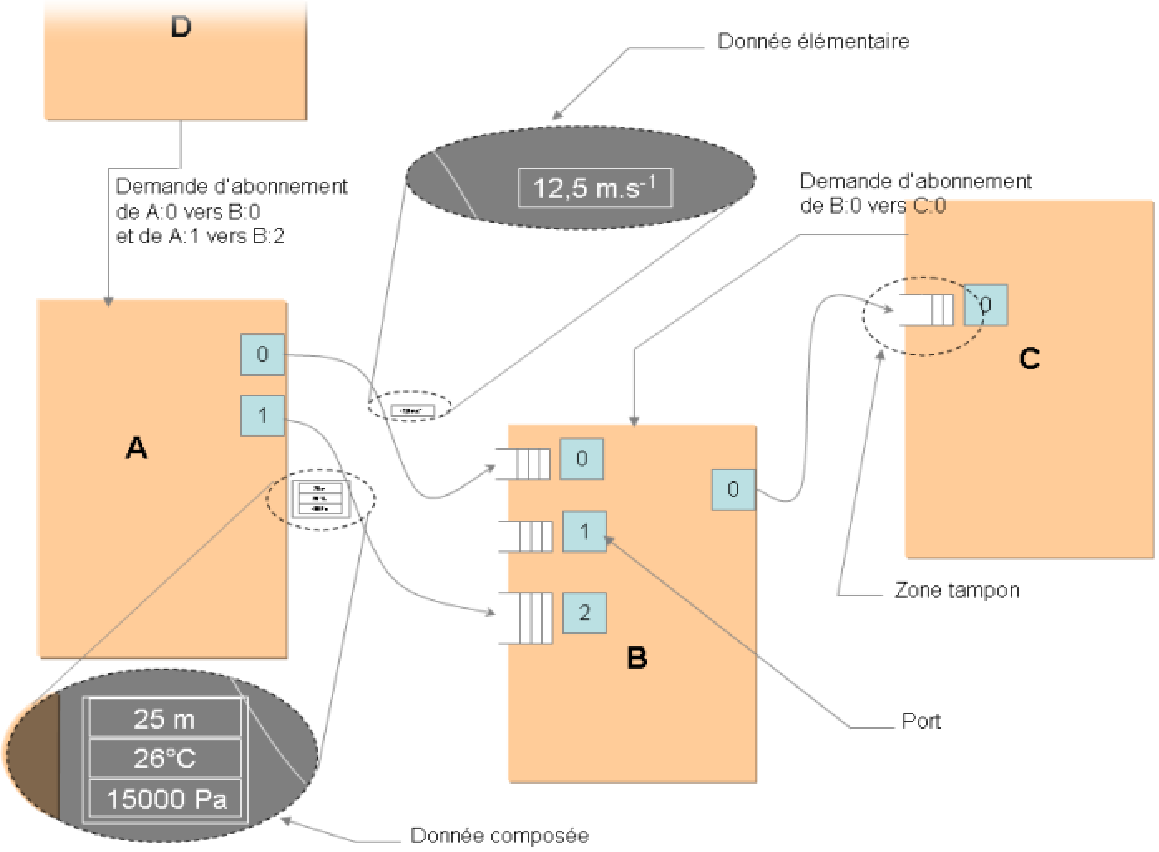

Figure 3 : Établissement de flux de données entre module

2.1.3) Décomposition logicielle d'une architecture de contrôle

A partir du modèle programmation et de l'organisation conceptuelle présentés précédemment nous avons définit une architecture de contrôle logicielle mettant en exergue la façon dont le logiciel est décomposé en terme de modules. Les principes de cette décomposition sont présentés de façon schématique dans le figure 4.

Tout d'abord nous supposons qu'un module est implémenté comme une **tâche temps-réel**, c'est-à-dire l'équivalent d'un processus pour un système d'exploitation temps-réel avec ordonnancement à priorités fixes (les priorités ne peuvent changer que sur requête de l'application temps-réel elle-même et non du fait de l'ordonnanceur de l'OS par exemple). Ceci implique qu'une implémentation correcte ne peut se faire que sur un système d'exploitation assumant cette caractéristique, ce qui est le cas de la plupart des systèmes temps-réel existants. En effet, cette caractéristique est essentielle à un fonctionnement conforme à notre spécification d'une architecture

conçue suivant la décomposition logicielle proposée. Nous supposons par ailleurs que les modules sont implémentés au sein de l'OS comme des **tâches asynchrones**, c'est-à-dire qu'elle ne seront pas réveillées périodiquement par l'ordonnanceur de l'OS à la différence de tâches périodiques. Grâce à ces deux caractéristiques nous pouvons (et nous devons) contrôler précisément les **moments d'activation** des modules (si nécessaire) et leurs **relations de préemption**. Au final cela nous permet d'avoir un contrôle complet et **déterministe** du système.

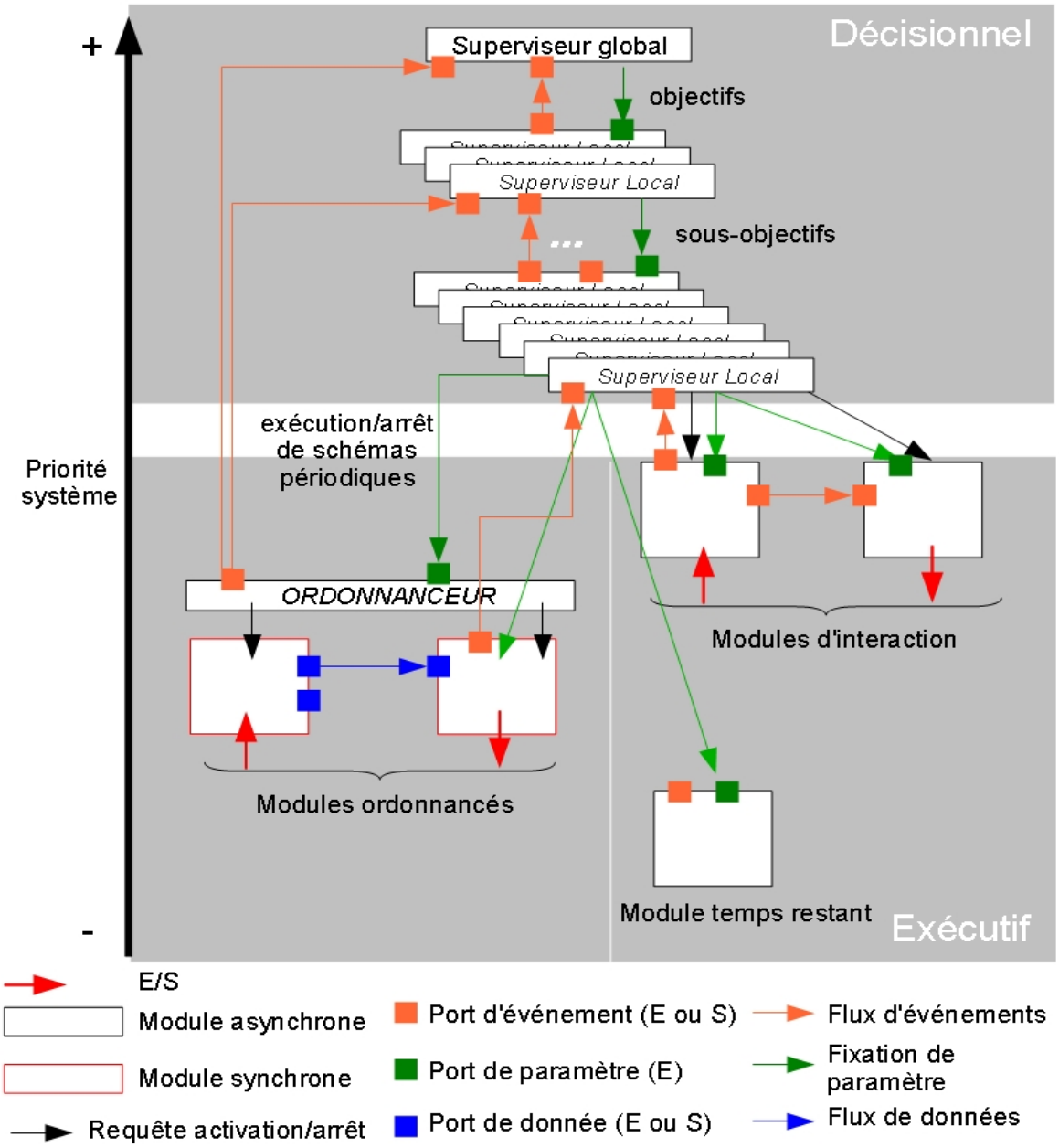

Figure 4 : Schéma résumant la décomposition logicielle proposée

A partir de ces premières propriétés la décomposition logicielle repose sur l'utilisation de modules pour implémenter l'ensemble des tâches robotiques. Ces modules sont conceptuellement hiérarchisés suivant leur priorité, de la plus forte (en haut dans la figure 4) à la plus faible (en bas dans la figure 4). Elle détaille l'organisation conceptuelle présentée dans la section 1.1.1 en décrivant les types de modules que contiennent chacune des deux couches et en définissant leurs relations possibles.

Dans la couche décisionnelle on retrouve l'ensemble des superviseurs servant à décomposer le mécanisme de prise de décision du robot. Les superviseurs suivent une relation hiérarchique depuis le **superviseur global** (unique point d'entrée du système chargé de gérer globalement l'application) jusqu'aux s**uperviseurs locaux** les plus proches (en terme de priorité) de la couche exécutive, cette relation hiérarchique déterminant leur **niveau de priorité** (au sens système d'exploitation et en même temps au sens décisionnel). Ces superviseurs, quel que soit leur niveau hiérarchique, peuvent interagir directement avec les superviseurs hiérarchiquement inférieurs ou avec les modules de la couche exécutive (voir plus loin). Les interactions entre superviseurs sont de deux types : événementielle et par fixation de paramètres. Les **interactions événementielles** permettent à un module superviseur de notifier le ou les superviseurs de niveau hiérarchique supérieur de l'occurrence d'un événement qu'il a détecté. Quel que soit son niveau, un superviseur **reçoit des paramètres qui correspondent aux objectifs** qu'il doit réaliser ou abandonner et **produit des paramètres qui correspondent à des sous-objectifs** qu'il souhaite voir réaliser ou abandonner. Cette organisation hiérarchique peut notamment servir à la gestion des modes : le superviseur global se charge de la **commutation des modes d'autonomie** alors que les superviseurs locaux peuvent représenter chacun **un mode d'autonomie particulier** (e.g. téléopération directe, télé-programmation, autonome, mixte, coopération multi-robots, etc.). Cette vision peut être raffinée autant de fois que nécessaire, par exemple en créant des modes plus ou moins dégradés des modes d'autonomie considérés en rajoutant un niveau hiérarchique de superviseurs locaux. La gestion de la préemption est simple : dès lors qu'un module superviseur est activé (sur réception d'un événement ou d'un paramètre) il préemptera tous les modules superviseurs de niveaux hiérarchiques (et donc de priorités, cf. fig. 4) inférieurs. Ceci se justifie par le fait que les décisions prises à un niveau décisionnel donné sont plus importantes (notamment parce qu'elles peuvent les remettre totalement en cause -i.e. déclencher l'abandon des objectifs) que les décisions prises dans les niveaux inférieurs et doivent dont être plus prioritaires.

Les modules superviseurs **partagent tous un même mécanisme de supervision**, configuré différemment en fonction de leur rôles respectifs. De façon schématique, nous considérons qu'un module superviseur définit un ensemble de **fonctions de supervision** (correspondant à un ensemble d'objectifs que peut réaliser le superviseur, de façon potentiellement concurrente), elles-même décrites à partir d'un **ensemble de règles**. Chaque **règle** est constituée :

- d'une **précondition**, qui détermine le contexte dans lequel la règle devient active si elle était inactive.
- d'un ensemble **d'actions**, qui sont exécutées dès lors que la règle devient active et qui sont annulées (si possible, -i.e. si ces actions sont structurelles) dès lors que la règle devient inactive.
- d'une **postcondition**, qui détermine le contexte dans lequel la règle devient inactive si elle était active.

L'idée directrice du mécanisme de supervision est que **dès qu'il y a un possible changement de contexte**, la précondition de chaque règle inactive ou la postcondition de chaque règle active est évaluée (uniquement pour les règles définissant les fonctions de supervision **actives**). Si elle est vraie alors la règle correspondante est soit activée, soit désactivée. Les **changements de contexte** utilisés au sein des pré et postconditions d'un superviseur donné sont notamment définis par l'**occurrence d'événements** provenant des superviseurs de niveau hiérarchique inférieur ou de la couche exécutive. Ils peuvent également être induits par l'occurrence d'événements liés au changement d'état interne au superviseur, c'est-à-dire à la **modification des règles actives et inactives ou à leur durée d'activité**, ce qui permet par exemple de gérer le séquencement de l'exécution de règles ou d'implémenter un comportement cyclique. Les conditions peuvent être exprimées par des alternatives (OU logique) sur l'occurrence d'événements simples ou de clauses d'événements (ET logique), ce qui permet d'exprimer des contextes détectés de façon plus ou moins complexe.

Parmi les actions possibles lors de l'activation d'une règles, deux catégories d'actions existent:

- Les **actions structurelles** qui consistent à activer et à mettre en relation des modules, sont de différents types : **activation d'un schéma périodique** (consiste à connecter les modules contenus dans le schéma et à les paramétrer puis à demander l'exécution du schéma à l'ordonnanceur, voir plus loin), **activation d'un schéma événementiel** (consiste à connecter les modules contenus dans le schéma, à les paramétrer puis à les activer directement), **activer une fonction de supervision** (envoi d'un paramètre au superviseur correspondant), **abonnement du superviseur à des événements** (produits par des modules de la couche exécutive ou des superviseurs de niveau hiérarchique inférieur), **activer un module temps-**restant (voir plus loin). Toutes ces actions structurelles sont annulées lorsque la règle qui les a lancée est désactivée.
- Les **actions spontanées** qui sont de plusieurs types : **effectuer un calcul court** (par exemple pour un calcul simple de fonction de jonction entre deux lois de commande), **notifier un événement** (aux superviseurs abonnés au port correspondant), **paramétrer un module synchrone ou un module d'interaction**. La caractéristique principale de ces actions est qu'elle ne peuvent être annulées lorsque la règle qui les a lancée est désactivée.

A partir de ce mécanisme générique pour mettre en œuvre une supervision « réactive » l'utilisateur ne devra donc plus que définir les fonctions de supervision de chaque superviseur, c'est-à-dire **les ensembles de règles activables et désactivables depuis le niveau hiérarchique supérieur**. Ces fonctions de supervision sont primordiales dans la définition de l'architecture de contrôle d'un robot puisqu'elle définissent sa structure dynamique (l'évolution au cours de la mission de l'interconnexion des modules de la couche exécutive et leur lien avec le décisionnel). Fournir un mécanisme générique pour la gestion de la supervision, est lié au fait que nous voulons imposer une « contrainte méthodologique », et ce pour deux raisons essentielles :

- Il s'agit de simplifier et de fiabiliser la tâche de l'architecte en lui fournissant tous les mécanismes de base pour gérer l'aspect dynamique de son architecture, l'ordonnancement et la supervision. Il n'a plus qu'à définir les fonctions de supervision et leur règles associées, l'exécution de ces règles étant assuré par le mécanisme générique.
- Il s'agit d'empêcher que l'architecte ne puisse définir par lui-même ses superviseurs, afin d'éviter des implémentations qui violerait le principe de réactivité des superviseurs (ou plus exactement de limiter celles-ci). En effet, si on suppose que la définition d'un superviseur est « libre » alors rien n'empêcherait l'architecte de définir des mécanisme de supervision

intégrant par exemple de la planification de mission à long terme, mécanisme utilisant des algorithmes couteux en temps qui pourraient **bloquer complètement le fonctionnement de la couche exécutive** (due aux relations de priorité, voir fin de cette section), ce qui reviendrait à perdre tout contrôle temps-réel sur le robot durant la planification.

Dans la **couche exécutive**, on retrouve globalement des modules se situant dans deux mondes différents un **monde continu** (en bas à gauche fig. 3) et un **monde événementiel** (en bas à droite fig. 3).

Pour gérer le monde continu, la décomposition logicielle définit qu'il y a un module **ordonnanceur** et des **modules synchrones**, ordonnancés par celui-ci. Rappelons que tous ces modules sont asynchrones au sens système, mais : le module ordonnanceur est **pseudo-périodique** (génère des timers pour se réveiller périodiquement, si besoin est en fonction du contexte) et les modules synchrones sont activés périodiquement par le module ordonnanceur qui leur envoi des requêtes d'activation (ils sont donc périodiques du point de vue applicatif). On suppose que le module ordonnanceur dispose d'informations relatives à l'ordonnancement des modules synchrones:

- leur **durée d'exécution nominale**, propre à chaque module.
- leur éventuelles **relations d'exclusion** pour les modules interagissant avec un système externe ou une entrée/sortie, qui sont définies pour l'ensemble des modules synchrones utilisés dans une application.
- les séquences de modules à activer, ainsi que leurs paramètres temporels (période et délai critiques), permettant de réaliser une action et/ou d'observer des phénomènes, qu'on regroupe dans des **schémas périodiques**. Chaque schéma périodique est utilisé dans un but applicatif donné, par exemple pour implanter une loi de commande, une observateur d'événements ou une chaine de diagnostics, ou tout cela en même temps.

Le module ordonnanceur est donc la pierre angulaire de cette décomposition logicielle car il est celui qui supporte le respect des contraintes temporelles (de la période d'exécution) ou critiques (non-blocage ou existence d'un module synchrone) essentielles au bon fonctionnement d'une application de contrôle temps-réel. Le fait de disposer d'un tel ordonnanceur applicatif permet de **contrôler** et d'**observer** de manière très fine le comportement du logiciel à l'exécution, caractéristique essentielle pour assurer une certaine fiabilité. Il est notamment capable de notifier des événements relatifs à des problèmes d'ordonnancement (retard considérés comme critiques) vers les superviseurs. Ces derniers activent le module ordonnanceur en lui demandant **d'activer ou de désactiver l'ordonnancement d'un ou plusieurs schémas** via l'envoi de paramètres. Notons que le module ordonnanceur est un module générique non modifiable par l'utilisateur mais paramétrable en fonction des informations relatives à l'ordonnancement (**schéma et modules connus**).

Les **modules synchrones** servent à implémenter les algorithmes et les entrée/sorties utilisés pour implanter les lois de commandes, observateurs, ou plus généralement, tout type de boucle de calcul et d'interaction périodique. Ils offrent la possibilité de décomposer à grain fin ces boucles afin de mieux en réutiliser différentes parties dans différents schémas. Ils peuvent donc interagir avec des entrées / sorties et/ou réaliser des calculs à partir de données en entrée et produisant des données en sortie. Usuellement on peut décomposer ces modules en différentes catégories (même si un même module doit pouvoir appartenir à plusieurs catégories) :

- Les **modules capteurs** servent à interroger les capteurs et à produire sur leur ports de sortie de données les données capteurs (normalisées) correspondantes.

- Les **modules actionneurs** servent à commander les actionneurs à partir des données décrivant les commandes présentes sur leurs ports d'entrée de données.
- Les **modules perception** servent à reconstituer des données complexes à partir de données issues de ses ports d'entrée de données et à produire ces données sur ses ports de sortie de données, ou le cas échéant à notifier des événements pertinent via ses ports de sortie d'événement.
- Les **modules action** servent, à partir de leurs données en entrée, à calculer et à produire les données correspondantes (via leurs ports de sortie de données) : des commandes pour des modules actionneurs ou des consignes pour d'autres modules action, ou le cas échéant de produire un événement pertinent relatif à ce calcul.
- Les **modules détection** s'ils servent à détecter certains dysfonctionnements algorithmiques ou matériels (i.e. fautes) dans l'architecture et à produire les indicateurs de ces fautes sur leurs ports de sortie de données, ou le cas échéant à générer un événement dans certains cas de figures.
- Les **modules diagnostic** servent à établir un diagnostic à partir de fautes détectées par les modules détection afin de générer des événements pertinents (cause du problème détecté par exemple) vers les superviseurs.

La possibilité de décomposer en grain fin les algorithmes utilisés en terme de composition de module, permet de laisser une flexibilité maximale à l'exécution : on peut par exemple dégrader une boucle de commande en simplifiant le traitement utilisé en remplaçant uniquement le module action et/ou le module perception, ou à l'inverse on peut complexifier cette boucle en lui ajoutant une chaine de diagnostic.

Les deux contraintes fortes sur les modules synchrones par rapport au modèle de programmation, est qu'ils ne peuvent **ni réagir à des événements** (i.e. ils ne peuvent déclarer de port d'entrée d'événement), **ni réagir à des modification de paramètres** (i.e. la modification de la valeur de leurs paramètres ne peut entrainer de réaction spécifique). Ces deux contraintes sont essentielles pour s'assurer de l'aspect synchrone de ces modules.

Pour gérer le monde événementiel, la décomposition définit qu'il y a deux types de modules : des **modules d'interaction** et des **modules temps-restant**. Ces modules sont des modules asynchrones, c'est-à-dire **non périodiques** et donc **non-ordonnancés par le module ordonnanceur** applicatif. Le principale contrainte sur ces modules synchrones est qu'ils ne peuvent communiquer via des flux de données (i.e. ils n'ont pas de port d'entrée / sortie de donnée).

Les **modules d'interaction** sont utilisés pour implanter des interactions avec des systèmes externes via des entrées / sorties qui ne sont pas temporellement prédictibles. Typiquement, dans un système téléopéré, ils servent à implanter les communications réseaux entre le poste de l'opérateur et le robot qui supportent les échanges au niveau décisionnel. En effet les échanges entre superviseurs ne sont pas temporellement prédictibles puisque par exemple l'opérateur peut décider à tout moment de donner ou modifier un ordre passé au robot ou dans le sens inverse le robot peut à tout moment notifier l'opérateur d'une situation pertinente. Les modules d'interaction peuvent également servir à implanter des interaction avec une interface graphique ou des périphériques événementiel (périphérique d'entrée utilisateur notamment) voire éventuellement avec une base de donnée temps-réel -i.e. à accès suffisamment rapide pour ne pas perturber le cycle temps-réel . Ces modules sont assemblés via leurs ports d'événement au sein de **schéma événementiels** qui seront

activés ou désactivés par les superviseurs. Certains de leurs ports de sortie d'événements sont utilisés par les superviseurs pour se voir notifier des événement considérés comme pertinents et leur ports de paramètres peuvent servir à paramétrer leur exécution « en ligne » et éventuellement à déclencher certaines réactions.

Les **modules temps restants** sont utilisés pour implanter des **calculs « longs »** que doivent effectués les superviseurs, comme par exemple des planifications de trajectoire, de chemin ou des planification de mission, voire des stockage de données considérés comme non « temps-réel » (dans une BDD à accès lent). L'idée majeure est de pouvoir déclencher des calculs sans que ceux-ci ne viennent perturber le cycle d'exécution temps-réels assuré par l'ordonnanceur : ils prendront le temps processeur laissé libre par le module ordonnanceur. Les modules temps-restant n'ont pas vocation à être assemblés avec d'autres modules : ils réalisent un calcul ou stockent une donnée sur réception d'une requête d'activation (éventuellement précédée d'une ou plusieurs requêtes de paramétrage pour les configurer) et notifient un événement contenant le résultat à la fin de leur calcul et éventuellement font plusieurs notifications intermédiaires si cela s'avère justifié (e.g. notification de l'avancement du calcul).

Si conceptuellement il n'y a pas de relation hiérarchique entre les modules de la couche exécutive, il y a néanmoins des relations de priorités entre ces différents types de modules pour assurer un fonctionnement temps-réel conforme :

- les **modules d'interaction sont les plus prioritaires** car ils sont les relais d'événements reçu par les superviseurs ou produit par ceux-ci à destination d'organes décisionnels (opérateur, autres superviseurs, etc.) et donc des relais de décision. Les changement de décision induit par ces événements peuvent remettre en cause tous les traitement utilisés à un moment donné dans la couche exécutive, y compris les modules d'interaction eux-mêmes (e.g. sur une perte de connexion d'un lien réseau).
- Le **module ordonnanceur** est de priorité **directement inférieure aux modules d'interaction** et **directement supérieure aux modules synchrone** qu'il ordonnance. En effet pour pouvoir ordonnancer les modules synchrones et le cas échéant préempter leur exécution, il doit leur être plus prioritaire.
- Les **modules synchrones** sont de priorité **inférieure à celle de l'ordonnanceur**. Ils ont deux niveaux de priorité suivant qu'ils sont en cours d'exécution (i.e. en cours d'utilisation du processeur – niveau haut) ou non (état d'attente ou état préempté, niveau bas), ce niveau étant fixé par l'ordonnanceur.
- Les **modules temps-restant** sont de priorité **inférieure au niveau bas des modules synchrone**, mais n'ont pas de priorité entre eux. C'est leur nature même qui explique cette priorité la plus basse dans l'architecture. Ils ne doivent pouvoir s'exécuter que si aucun module synchrone n'a requis le processeur.

L'articulation entre les deux niveaux est relativement simple **le niveau décisionnel est plus prioritaire que le niveau exécutif**. Ainsi les superviseurs seront toujours plus prioritaires que n'importe quel module de la couche exécutive. C'est le principe de prépondérance de la réaction du décisionnel qui s'applique, car les décisions peuvent remettre en cause tous les calcul et interactions effectuées dans la couche exécutive.

Globalement cette décomposition logicielle occulte les problèmes liés à la gestion de la préemption, puisque celle-ci est systématisée dans l'architecture, ce qui permet à l'utilisateur d'avoir

une vision simplifiée et unifiée de la préemption.

2.1.4) Remarques et conclusion sur l'architecture générique

L'étude (parallèle) des besoins technique pour les architectures de contrôle des robots PROSIT et ASSIST a joué un rôle important dans l'élaboration de la proposition d'architecture ContrACT comme évolution de l'architecture COTAMA ***[Eljalaoui 2007]***. La problématique majeure, du point de la conception logicielle, vient de la gestion du lien de téléopération supervisé et de la possibilité d'intégrer des mécanismes délibératifs à temps-non contraints. Ces études ont permis d'enrichir .

Premièrement, il fallait qu'existent des modules capables d' « écouter » de manière bloquante (attente passive) le réseau pour permettre la mise en place d'un lien de communication en mode connecté entre le poste opérateur et le robot. Ce besoin est notamment apparu dans le contexte du projet PROSIT. Cette attente bloquante est impérative afin que le robot reste en permanence à l'écoute d'ordres en provenance du poste opérateur ou le cas échéant de détecter un problème de connexion réseau avec le poste opérateur. De manière symétrique, puisque le poste opérateur devait disposer de sa propre architecture de contrôle (car il embarque également des boucles de commande pour gérer par exemple le retour de force), il devait également pouvoir attendre des notification de situation (problèmes rencontrés) de la part du robot. L'utilisation d'un protocole réseau en mode connecté est essentielle pour les échanges de messages entre superviseurs distants : ceux-ci doivent être notifier dès lors que la connexion est rompue (pour y réagir, par exemple par une commutation du mode téléopéré vers un mode autonome pour le robot) ; les messages échangés sont temporellement imprévisibles et une absence de message ne peut être considérée comme une erreur. Considérant nos travaux antérieurs il était impossible d'envisager d'utiliser le TCP (ou tout autre protocole réseau de type connecté) au sein de modules ordonnancés, à part dans certaines conditions très spécifiques ne pouvant certainement pas être respectées dans le cas de communications longue distance à travers l'internet ou utilisant des dispositifs très faibles en débit (communication via satellite). C'est ainsi qu'est venue l'idée d'ajouter à la couche exécutive des modules asynchrones pour gérer les interactions de niveau « supervision » entre architectures de contrôle distantes. Cette idée a été dans un deuxième temps renforcée par la nécessaire interaction entre l'architecture de contrôle temps-réel du poste opérateur et l'interface graphique utilisée par l'opérateur pour interagir avec le système. En effet, dans ce cas on retrouve les mêmes impossibilité à prévoir temporellement les communication entre IHM et architecture de contrôle. En généralisant le principe d'interaction non pas entre uniquement entre architectures distantes mais entre une architecture n'importe quel type de système externe (notamment des IHM), le principe de **module asynchrone d'interaction** a montré sa pertinence pour un ensemble de problématiques proches, liés au besoin de **nourrir le niveau décisionnel d'événements ne venant pas uniquement d'un monde continu mais aussi d'un monde événementiel**.

Deuxièmement, la gestion de la supervision a demandé à **repenser complètement les mécanismes de supervision** et dans une moindre mesure le modèle de programmation. Il fallait notamment que des données et non plus seulement des signaux puissent transiter avec les événements de manière à ce que les superviseurs aient des données sur lesquelles travailler et à partir desquelles prendre des décisions. Par exemple, un événement (provenant du poste opérateur) représentant un changement de consigne devait pouvoir contenir ladite consigne afin que le superviseur concerné puisse configurer de manière adéquate le schéma de commande périodique.

Cela a demandé de modifier le modèle de programmation afin que les événements soient associés à des données. Cela requiert également de repenser les superviseurs afin qu'ils puissent gérer ces données (extraction de la donnée d'un événement, paramétrage contextuel de schémas), ce qui a amené un travail important sur le middleware robotique (voir section 2.2). La nécessité de sélectivité des événements a également été induite par le lien de téléopération : nous voulions que certaines données associées aux événements puisse représenter des messages « haut-niveau » comme des changement de mode et que l'on puisse sélectionner les actions des superviseurs en fonction des valeurs de ces données, afin notamment d'éviter l'explosion du nombre de signaux (i.e. types d'événements possibles impactant directement sur le nombre de ports d'entrée d'événements des superviseurs) et de simplifier la « programmation » des superviseurs. Le cas typique est par exemple un événement stipulant une demande de changement de mode et dont la donnée représente le nouveau mode à activer : il fallait qu'en fonction de la donnée nous puissions activer le mode (i.e. activer le superviseur) correspondant. L'ensemble des modifications envisagées dans le mécanisme de supervision et la manière de le configurer devenue beaucoup plus « riche » a fait émerger l'idée d'un langage de programmation des superviseurs, comme nous le verrons dans les section 2.2 et 2.3.

Enfin, il fallait pouvoir intégrer conjointement des **mécanismes délibératifs** à l'architecture de contrôle, ce qui était impossible auparavant dans l'architecture COTAMA. Ce besoin est notamment apparu dans le contexte du projet ASSIST, dans lequel le robot doit intégrer des mécanismes de planification de mission. Pour résoudre ce problème, l'idée a été d'exploiter des modules asynchrones particuliers s'exécutant de manière moins prioritaires que les modules synchrones. De par cette simple propriété, ces modules s'exécutent en temps-restant par rapport aux modules synchrones couramment ordonnancés et plus généralement par rapport à tous les autres modules. De fait l'exécution de ces modules, souvent très gourmand en ressources processeur, ne viendra **pas perturber les cycles d'exécution temps-réel**.

2.2) Middleware Robotique

Le middleware robotique est un terme utilisé pour regrouper un ensemble d'artefacts logiciels qui constituent une plateforme logicielle générique, utilisée pour développer des architectures de contrôle logicielles suivant la décomposition présentée en section 2.1.3. Il s'agit de fournir aux développeurs les moyens de base leur permettant de se focaliser sur le code applicatif (à opposer au code « système » qui est générique) mais aussi de faciliter le portage des applications sur différents systèmes d'exploitation temps-réel.

Les artefacts constituant le middleware robotique, présentés dans les sous-section suivantes, sont:

1. Une **librairie** permettant de programmer des **modules** suivant le modèle de programmation proposé en section 2.1.2, fournissant une interface générique et des implémentations dépendantes du système d'exploitation temps-réel utilisé.
2. Des squelettes **de code** pour les différents types de modules rencontrés.
3. Une implémentation des **mécanismes génériques** en jeu dans l'architecture, c'est-à-dire les mécanismes d'ordonnancement et de supervision.

2.2.1 La librairie libmodule

La librairie C *libmodule* a été développée pour implémenter le modèle de programmation proposé en section 2.1.2. Sa première version, qui a largement inspiré la version actuelle, a été développée au LIRMM en 2006 pour l'implémentation d'architectures de contrôle de robot sous-marin ***[Trapier 2006]***. Elle est vue comme une abstraction du système d'exploitation temps-réel utilisé, ce qui permet à l'utilisateur de définir des modules potentiellement indépendants (si par ailleurs le code de ces modules n'appelle pas d'autres primitives systèmes que celles fournies par la librairie) d'un système d'exploitation donné. Ceci est une qualité essentielle si on considère la possibilité de pouvoir facilement « porter » le code applicatif sur un autre système d'exploitation temps-réel. En effet, si les développeurs respectent au l'idée de n'utiliser que des primitives offertes par la librairie, ainsi que des appels à des librairies standard (e.g. posix), le seul travail de portage à faire se situera au niveau du développement d'une nouvelle implémentation de la librairie pour le nouvel OS temps-réel considéré. A l'heure actuelle, nous n'avons fourni une implémentation de *libmodule* pour le système temps-réel linux RTAI (real time application interface), un des systèmes temps-réel les plus populaires, soutenu par une communauté très active. Cette implémentation sert d'implémentation de référence pour *libmodule*.

Dans cette implémentation de référence, un module est une tâche temps réel. Une tâche temps réel a la spécificité d'être **plus prioritaire que le noyau du système d'exploitation non-temps réel** (dans notre cas linux) ou à défaut que **plus prioritaire que tous les processus** exécuté par ce système d'exploitation non-temps-réel (dans notre cas les processus linux). Dans RTAI, il existe deux possibilités pour implémenter des tâches temps-réel : comme des **modules noyau** ou comme des **tâches LXRT**. Nous avons choisi de les implémenter comme des tâches LXRT pour la souplesse qu'offre l'API LXRT : le code d'une tâche LXRT est créé comme pour un processus classique à la différence que certains appels à l'API permettent de rendre cette tâche temps-réel et d'effectuer des communications dans un monde temps-réel (i.e. sans passer par le noyau linux). Au delà de la simplicité de programmation, l'autre énorme avantage de LXRT (par rapport à des modules noyau) est qu'elle laisse néanmoins la possibilité à une tâche d'effectuer des appels au noyau linux, ce qui s'avère très utile pour l'utilisation de drivers et de librairies diverses. Dans ce cas la tâche passe en mode « temps-réel mou » ce qui signifie quelle devient moins prioritaire que les tâches « temps-réel dur » et que le noyau linux mais reste plus prioritaire que tous les processus linux. Afin de préserver les relations de priorité d'exécution, essentielles dans l'architecture de contrôle proposée, nous devons nous assurer que les modules, une fois leur phase d'initialisation terminée, ne sortiront plus du mode « temps-réel dur ». Pour cela nous devons limiter au maximum les appels au noyau linux :

- Il faut interdire l'allocation dynamique de mémoire après l'initialisation, car celle-ci provoque un basculement en mode « temps réel mou ».
- Il faut éviter autant que possible des appels directs aux drivers linux (dans les modules de la couche exécutive) en utilisant des serveurs linux, qui sont des tâches spécifiques auxquelles sont déléguées les appels au noyau linux.
- Si des threads doivent être utilisés, il ne faut utiliser que des threads temps-réel (rendu « temps-réel dur » de la même façon que les tâches LXRT) et leur création ne peut se faire qu'à l'initialisation des modules qui les créent.

Ces sécurités vont permettre d'éviter de modifier le mode d'exécution des modules (i.e. que les modules s'exécutent en mode « temps-réel mou » le temps d'appels système) et ainsi s'assurer le respect des principes architecturaux pour ce qui concerne le respect des priorités des modules.

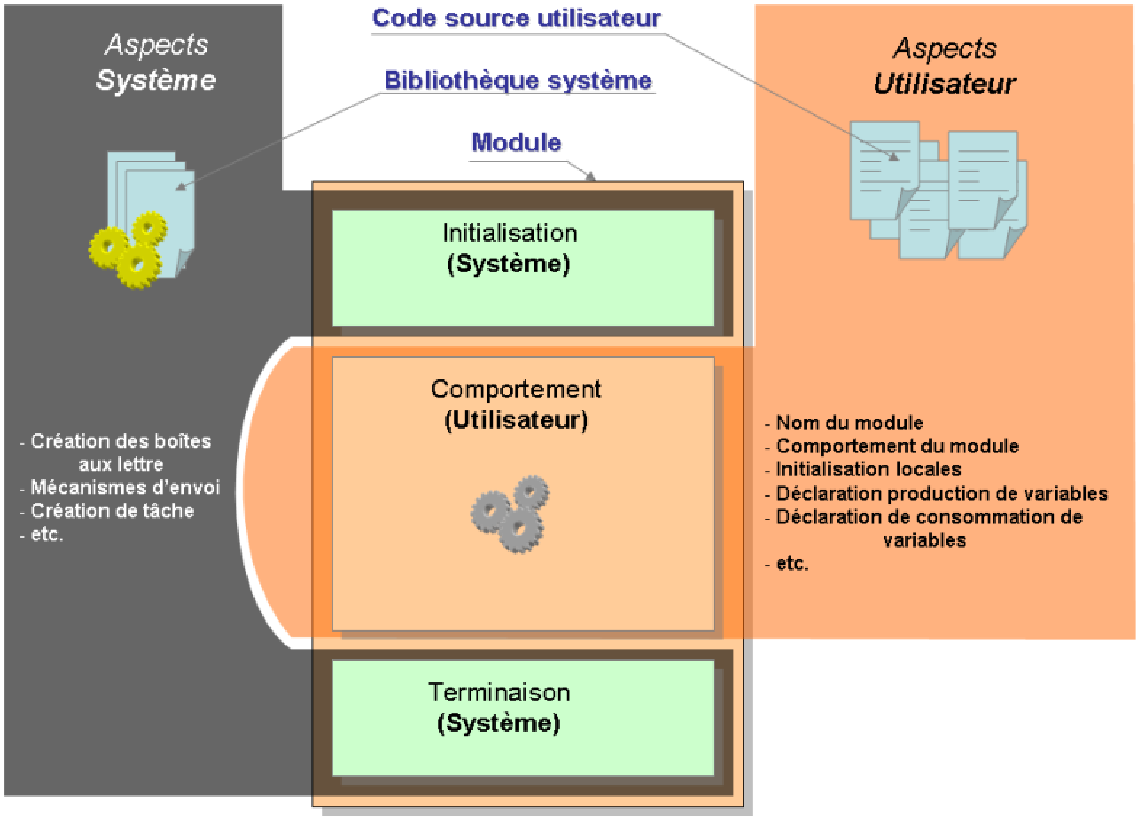

Figure 5: articulation entre aspects système et utilisateur pour l'implémentation d'un module

La librairies définie des primitives et des structures de données *internes* et *utilisateur*. La figure 5 donne une représentation schématique de l'articulation entre les aspects génériques (i.e. sur couche système) et les aspects utilisateurs, c'est à dire le code définit par l'utilisateur à partir des primitives utilisateurs et éventuellement d'autres librairies.

Les structures de données et primitives *internes* servent à gérer l'ensemble des mécanismes de communication des modules, c'est-à-dire leurs différents ports (requête, paramètres, données et événements). Les *primitives internes* sont appelées par la fonction `main` à l'**initialisation** et à la **terminaison** du module et servent essentiellement à initialiser/détruire les *structures de données internes* à partir des *structures de données utilisateur* (voir plus loin). La fonction `main` du module est donc globalement charger de créer le module et d'en faire une tâche temps-réel (LXRT). Cette fonction `main` **n'est pas modifiable par l'utilisateur**. Le code utilisateur (cf. fig. 5) programmant l'interface et le comportement du module est regroupé au sein d'une fonction secondaire, nommée `ModuleMain` (cf. fig.6), qui est elle-même appelée par la fonction `main`. Grâce à cela, l'utilisateur ne peux pas à accéder au code propre à l'implémentation de la librairie sur l'OS considéré (RTAI), contenu dans la fonction `main`.

L'utilisateur doit donc programmer le module à partir des *structures de données et des primitives utilisateur* qui lui sont proposées par la librairie. Les *structures de données utilisateur* servent à définir l'interface du module (cf. fig. 6), c'est-à-dire :

- son **nom** (cf. `MODULE_NAME`, fig. 6), qui sert d'identifiant unique dans le système (l'utilisateur doit s'assurer qu'il soit unique) pour identifier le module.
- sa **priorité** système (cf. `MODULE_PRIORITY`, fig. 6), qui permet de mémoriser la priorité nominale du module (afin de lui rendre cette priorité après des modifications réalisées de manière opaque par la librairie).
- son **auteur** (inutile pour le bon fonctionnement du module), qui identifie le ou les programmeurs du module.
- sa **description** (inutile pour le bon fonctionnement du module), qui donne un résumé des propriétés et de l'utilité principales du module.
- ses **ports de données en entrée** (cf. tableau `IUSE`, fig. 6), identifiés par un **numéro de port**, l'adresse de la **variable** (C) dans laquelle est stockée la donnée reçue et la **taille** de cette variable.
- ses **ports de données en sortie** (cf. tableau `IPRODUCE`, fig. 6), identifiés par un **numéro de port**, l'adresse de la **variable** (C) dans laquelle est stockée la donnée émise et la **taille** de cette variable. A ces **informations obligatoires** peuvent être ajoutées des **informations de connexion de ce port**, il faut alors indiquer le module producteur, son port et sa périodicité d'émission.
- ses **ports d'événements en entrée** (cf. tableau `IREACT`, fig. 6), identifiés par un **numéro de port**, l'adresse de la **variable** (C) dans laquelle est stockée la donnée de l'événement reçu et la **taille** de cette variable. A ces **informations obligatoires** peuvent être ajoutées des **informations de connexion de ce port**, il faut alors indiquer le module producteur, son port et son mode de production (oneshot ou continu).
- ses **ports d'événements en sortie** (cf. tableau `IDETECT`, fig. 6), identifiés par un **numéro de port**, l'adresse de la **variable** (C) dans laquelle est stockée la donnée de l'événement notifié et la **taille** de cette variable.
- le tableau contenant les tailles de tous ses **paramètres en entrée** (cf. tableau `PARAM_SIZE_IN`, fig. 6), c'est-à-dire paramètres susceptibles d'être modifiés par des requêtes de configuration reçues par le module.
- le tableau contenant les tailles de tous ses **paramètres en sortie** (cf. tableau `PARAM_SIZE_OUT`, fig. 6), c'est-à-dire les paramètres d'autre modules que le module est susceptible de modifier pendant son exécution via envoie de requête de configuration.

Notons que nous n'utilisons pas, par défaut, les informations de connexion associées au ports d'entrée (de données et d'événement) car elles obligent à des connexions prédéfinies ce que viole en le principe de dynamicité des connexions et couple trop fortement des modules, réduisant ainsi leur réutilisation. Néanmoins nous avons conservé ce mécanisme qui peut s'avérer utile pour des considérations de débogage (e.g. production de données utiles pour un module chargé du logging dans l'application).

Comme stipulé précédemment il faut associer des variables à tous les ports. Ces variables doivent exister tout au long de la vie du module, c'est-à-dire du point de vue utilisateur avant et après l'appel à la fonction `ModuleMain`. Nous utilisons donc pour cela des variables globales. Ces variables serviront à tous les appels de *primitives utilisateur* qui exploitent les ports.

Les *primitives utilisateur* fournies par la librairie permettent essentiellement à l'utilisateur de d'exploiter les moyens d'interaction des modules, autrement dit d'utiliser leurs ports. Le **port de**

requêtes d'un module correspond au point d'entrée global du module grâce auquel seront reçus tous les messages requérant une activation du module. Ces messages sont de différents types:

- L'**activation** et l'**arrêt** de son activité nominale.
- La demande de **terminaison** (entrainant la disparition du module dans le système).
- Des **acquittement** d'exécution ou de réception de requêtes d'activation/arrêt/terminaison.
- La **configuration** de ses paramètres.
- La **notification** de l'occurrence d'événements.
- L'**abonnement** et le **désabonnement** de modules à ses ports de sortie de **données** et d'**événements**.

```
...

// information on module

MODULE_DESCRIPTION("say what the module does");
MODULE_AUTHOR("Your name");

//Exchanges Management Variables
int input, output;
float notify, react;

// module name
char MODULE_NAME[] = "MOD";
//module priority
int MODULE_PRIORITY = 37;

// input data flows : example = { &Variable, PortReception, Size,
Periodicity (important if module name is set), Name of emitter module(can
be unused by setting value to "---"), PortModuleEmetteur} ou { &Variable,
PortReception, Size, 0, "---", 0}
ModuleUse IUSE[] = { {&input, 0, sizeof(int), 0, "---", 0}, IUSE_TERM };

// output data flows : example = { &Variable, PortEmision, Size}
ModuleProduce IPRODUCE[] = { {&output, 0, sizeof(int)}, IPRODUCE_TERM };

//input event flows : example = { &Variable, PortReception, size, module
name (can be unused by setting value to "---"), PortEmision, oneshot}
ModuleReact IREACT [] = { {&react, 0, sizeof(float), "---", 0, 0},
IREACT_TERM };

//output event flows : example = { &variable, PortEmision, size }
ModuleDetect IDETECT [] = { {&notify, 0, sizeof(float)}, IDETECT_TERM };

// size of parameters that can be received in the request mailbox:
size_t PARAM_SIZE_IN[] = { sizeof(double), PARAM_SIZE_TERM };

// size of parameters or events that can be sent in request mailboxes of
other modules :

size_t PARAM_SIZE_OUT[] = {sizeof(char)*256, PARAM_SIZE_TERM };

// description of the module behavior

...
int ModuleMain(int argc, const char * argv[]) {
...
}
```

Figure 6: exemple des structures de donnée utilisateur d'un module

Ce port est associé à une boite aux lettre système, sur laquelle le module vient lire les requêtes correspondantes, via deux primitives utilisateur, l'une bloquante (`WaitForRequest`) et l'autre non bloquante (`LookForRequest`). Ces deux primitives permettent d'extraire de la boite aux lettre des données de type `RequestMessage`, représentant à la fois le type de message (cf. paragraphe précédent) et les éventuels arguments associés, par exemple:

- L'identifiant acquittement, demandé pour une requête d'activation/arrêt ou retourné pour une requête d'acquittement.
- Les couples modules et ports consommateurs ainsi que le port producteur cible pour des requêtes d'abonnement et de désabonnement.
- Le port de paramètre cible pour une requête de configuration.
- Le port d'entrée d'événements cible pour une requête de notification.
- Etc.

Les deux primitives de réception de requêtes vont également extraire de la boite aux lettre globale les **données** associées aux requêtes de **configuration de paramètres** et aux requêtes de **notification d'événements**, et effectuer automatiquement des opérations spécifiques:

- La donnée associée à la **requête de configuration** est stockée dans un tampon interne. Elle sera affectée à une variable (et retirée du tampon) au moment de l'appel à la fonction `GetDataFromSetParamRequest`.(qui par défaut devrait suivre immédiatement la réception de la requête).

- La donnée associée à la **requête de notification** est automatiquement affectée à la variable associé au port d'entrée d'événement ciblé par la requête. La fonction `GetEventId` (qui par défaut devrait suivre immédiatement la réception de la requête) permet de connaître le port ciblé.

L'appel aux primitives d'abonnement et de désabonnement de données (respectivement `SubscribeUser` et `UnsubscribeUser`) et d'événements (respectivement `SubscribeUserToEvent` et `UnsubscribeUserToEvent`) produits par le module devraient quant à elle être par défaut appelées immédiatement après la réception de la requête d'abonnement ou de désabonnement (elle utilisent notamment des informations contenus dans ces requêtes).

De manière symétrique aux primitives utilisés pour la réception de requêtes il existe des primitives pour émettre des requêtes vers d'autres modules:

- `SubscribeContextualVariable` (resp. `SubscribeVariables`) pour souscrire un port (resp. des ports initialement connectés) d'entrée de données d'un module à un port de sortie de donnée (resp. aux ports producteurs de données associés aux ports initialement connectés) d'un autre module. `UnsubscribeContextualVariable` (resp. `UnsubscribeVariables`) réalise l'opération inverse.
- `SubscribeContextualEvent` (resp. `SubscribeEvent`) pour souscrire un port (resp. des ports initialement connectés) d'entrée d'événements d'un module à un port de sortie de d'événement (resp. aux ports producteurs d'événement associés aux ports initialement connectés) d'un autre module. `UnsubscribeContextualEvent` (resp. `UnsubscribeEvent`) réalise l'opération inverse.
- `SendRequest` pour émettre une requête d'activation ou d'arrêt.

- `SendAck` pour émettre un acquittement à une requête d'activation/arrêt.
- `PostEvent` pour notifier un événement.
- `SetModuleParam` pour émettre une requête de configuration.

Les modules peuvent aussi utiliser des communication de type « flux de données », pour cela le module possède **une boite à lettre pour chaque port d'entrée de données**. Ces boites à lettres sont créées automatiquement par les *primitives internes* à partir des informations contenues dans la structure `IREACT` (fig. 6). La gestion des ports de données (gestion de l'abonnement et de la mise à jour des variables) est effectuée par des primitives internes qui exploitent des structures de données internes, qui représentent par exemple la mémorisation des abonnés (c'est aussi le cas pour les ports d'événements). Des *primitives utilisateurs* permettent à un module de produire une donnée sur un port de sortie (`PostData`), ce qui revient à écrire dans la boite aux lettre d'entrée de donnée d'un autre module, ou de consommer une donnée sur un port d'entrée (`GetData` pour obtenir la première donnée disponible et `GetLatestData` pour obtenir la donnée disponible la plus à jour), ce qui revient à lire une donnée présente dans une de ses boites aux lettre d'entrée de données.

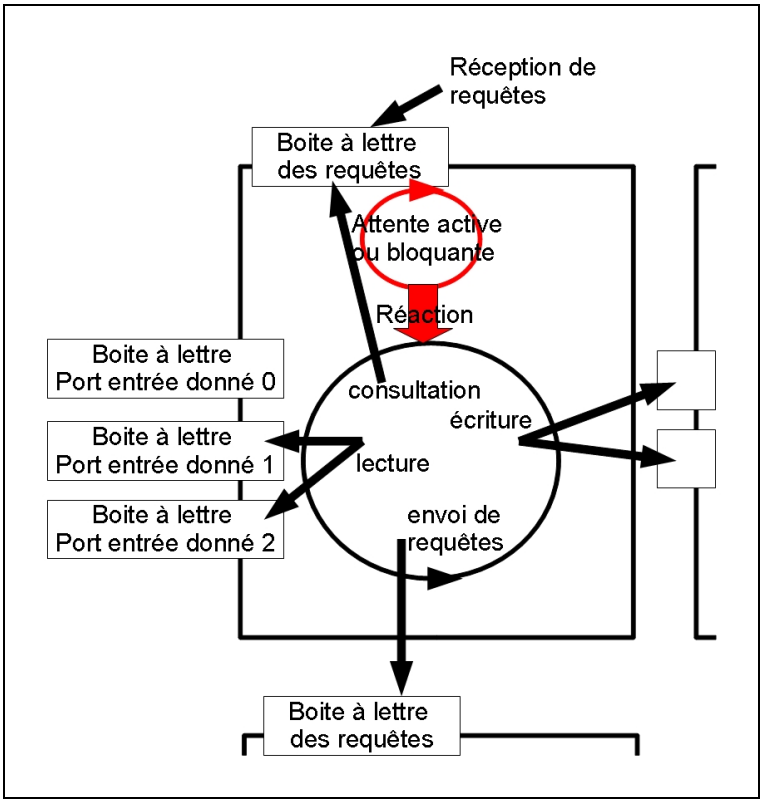

Figure 7 : représentation schématique du fonctionnement d'un module

Pour résumer cette section, le fonctionnement global d'un module est schématisé dans la figure 7. Un module possède une boite à lettre de requêtes sur laquelle il procède en interne à une attente active ou bloquante. Dès qu'il reçoit une requête dans cette boite à lettre et que l'ordonnanceur système lui donne le droit de s'exécuter il y réagit en exécutant un comportement qui est implémenté à partir d'opérations internes (par exemple la réalisation d'une demande

d'abonnement), d'opération de lecture (non bloquante) de ses boites aux lettre d'entrée de donnée (e.g. `GetData`) ou de requête (e.g. `LookForRequest`) ou d'interactions avec d'autres modules. Dans ce dernier cas le module peut soit écrire dans des boites aux lettres correspondant à des ports d'entrée de données abonnés à ses propres port de sortie de données (`PostData`), auquel cas il ne provoquera pas de réaction de ces modules, soit émettre des requêtes, vers d'autres modules, auquel cas il provoquera une réaction de leur part dès que l'ordonnanceur système leur donnera la main.

2.2.2 Squelettes de code des modules

A partir de la librairie *libmodule*, l'utilisateur peut définir différentes sortes de modules suivant la manière dont il définit les comportements de ces modules (i.e. dont il implémente la fonction `ModuleMain`, cf. fig.6). Les modules fonctionneront de manières très différentes en fonction de la manière dont l'utilisateur articule les appels aux primitives fournies par libmodule. Il peut par exemple utiliser une boucle d'attente active temporisée ou bloquante (cf. fig. 7) puis faire une sélection sur les messages reçus et réagir en conséquence. Il peut également faire en sorte que son module n'attende qu'une seule activation avant de se terminer. Suivant la solution choisie, le comportement du système varie fortement.

Si cette liberté est une bonne chose pour permettre de réutiliser la librairie *libmodule* dans d'autres contextes que celui exposé dans ce document (on peut imaginer proposer des architectures logicielles temps-réel orientées flux de données qui soient implantée avec cette librairie), il est nécessaire de fournir une solution compatible avec la décomposition logicielle proposée. Pour cela, nous devons imposer certaines propriétés aux modules, que nous concrétisons sous la forme de **squelettes de code**.

Nous avons tout d'abord défini un squelette de module commun à tous les types de modules potentiellement présents dans une architecture de contrôle telle que nous l'avons décrite dans la section 2.1 et que nous spécialisons par la suite pour chaque type de module. Ce squelette commun, présenté dans la figure 8 décrit la boucle d'attente et de réaction d'un module quelconque telle (schématisée dans la figure 7) en limitant les choix possibles dans l'utilisation des primitives fournies par la librairie *libmodule*. Ceci dans le but de fournir un comportement global aux modules qui soit conforme à la décomposition logicielle proposée en section 2.1.3.

Le comportement global d'un module quelconque est implémenté par une **boucle d'attente bloquante interruptible** sur sa boite de requêtes. Le choix d'une boucle d'attente bloquante est essentiel pour le bon fonctionnement de l'architecture telle que nous l'avons défini dans la section 2.1 car :

- Le module doit **rester en permanence réactif** à des requêtes car celles-ci peuvent arriver à n'importe quel moment de leur exécution. Il est donc nécessaire qu'il boucle en permanence sur sa boite de requêtes car il est impossible, dans l'architecture générique telle que nous l'avons défini, des séquences de réception de requêtes prédéfinies. Il serait éventuellement possible de définir de telles séquence pour une application données mais cela reviendrait à redéfinir en partie le middleware pour chaque application, ce qui est évidemment exclu car allant à l'encontre même du principe de middleware. Cette boucle d'attente « permanente » est simplement codée (cf. fig. 8) sous la forme d'un `while` dans laquelle s'effectue la sélection de la réaction à appliquer (attente si le module n'a rien à faire, ou action spécifique

en réaction à la réception d'un message) elle-même représentée par un `switch` sur le type du message reçu représenté par la variable `receivedMessage`.

- Le module doit pouvoir être **terminable** c'est-à-dire qu'il doit pouvoir effectivement être détruit à partir d'une requête provenant d'un autre module ou d'une commande utilisateur. Ce mécanisme est surtout utilisé pour des raisons de sécurité. Cette caractéristique est simplement codée via l'utilisation d'une variable (`end`, fig. 8) servant de condition de terminaison (`faux` par défaut, fig. 8) à la boucle d'attente. Lors de la réception de la requête de terminaison (`REQ_KILL`, fig. 8) cette variable est mise à `vrai` ce qui entraine la terminaison effective de la boucle d'attente et plus généralement du module.

- Le module **ne doit pas consommer du temps processeur** pendant qu'il attend des messages, l'utilisation d'une boucle d'attente active est donc impossible. Ceci se justifie par le fait, par exemple, qu'il serait impossible d'ordonnancer correctement les modules synchrones si ceux-ci consommaient du temps processeur de façon incontrôlable par le module ordonnanceur. D'autre part les superviseurs ne cesseraient dans ce cas de préempter les modules de la couche exécutive sans que cela soit nécessaire, ce qui remettrait en cause le fonctionnement du système en temps-réel. De manière générale, au delà d'une diminution des performances, ce sont la contrôlabilité et la gestion de la préemption qui seraient remises en cause. L'attente bloquante est implémentée par l'utilisation de la primitive `WaitForRequest` dès lors que le module n'a plus de traitement à effectuer en réaction à la réception d'un message (cas `REQ_UNKNOWN` exprimant qu'il n'y a plus de réaction à effectuer, fig. 8).

```
int ModuleMain(int argc, const char * argv[]){

RequestMessage receivedMessage;//variable containg the current received message
int end=0;
//initializing the module
receivedMessage.Type=REQ_UNKNOWN;//no message received by default
SubscribeVariables();SubscribeEvent();//connecting initially connected ports

while(!end)//request messages reception loop
{
      switch(receivedMessage.Type){
      case REQ_UNKNOWN://when nothing to do
            WaitForRequest(&receivedMessage);//waiting for a message
            break;
      case REQ_START://starting nominal activity of the module
            ...
            receivedMessage.Type=REQ_UNKNOWN;
            break;

      case REQ_STOP://stopping nominal activity of the module
            ...
            receivedMessage.Type=REQ_UNKNOWN;
            break;

      case REQ_KILL://terminating the module
            end = 1;
            receivedMessage.Type=REQ_UNKNOWN;
            break;

      case REQ_ACK://receiving an acknowledge from a module
            ...
            receivedMessage.Type=REQ_UNKNOWN;
            break;

      case REQ_EVENT://reacting to an event
            switch(GetEventId()){/*doing some stuff depending on the event*/}
            receivedMessage.Type=REQ_UNKNOWN;
            break;

      case REQ_SETPARAM://reaction to a configuration request
            switch(receivedMessage.Param.SetParam.Port){
            /*doing some stuff depending on the param using GetDataFromSetParamRequest*/}
            receivedMessage.Type=REQ_UNKNOWN;
            break;

      case REQ_SUBSCRIBE://reaction to a produced variable subscription request
            SubscribeUser(receivedMessage.Param.Subscribe.SourcePort,
receivedMessage.Param.Subscribe.TargetModule, receivedMessage.Param.Subscribe.TargetPort,
receivedMessage.Param.Subscribe.Periodicity,receivedMessage.Param.Subscribe.isContextual);
            receivedMessage.Type=REQ_UNKNOWN;
            break;

      case REQ_UNSUBSCRIBE://reaction to a produced variable unsubscription request
            UnsubscribeUser(receivedMessage.Param.Unsubscribe.SourcePort,
receivedMessage.Param.Unsubscribe.TargetModule, receivedMessage.Param.Unsubscribe.TargetPort);
            receivedMessage.Type=REQ_UNKNOWN;
            break;

      case REQ_SUB_EVENT://reaction to a produced event subscription request
            SubscribeUserToEvent(receivedMessage.Param.SubscribeEvent.targetModule,
receivedMessage.Param.SubscribeEvent.targetPort, receivedMessage.Param.SubscribeEvent.sourcePort,
receivedMessage.Param.SubscribeEvent.mode);
            receivedMessage.Type=REQ_UNKNOWN;
            break;

      case REQ_UNSUB_EVENT://reaction to a produced event unsubscription request
            UnsubscribeUserToEvent(receivedMessage.Param.UnsubscribeEvent.targetModule,
receivedMessage.Param.UnsubscribeEvent.targetPort,
receivedMessage.Param.UnsubscribeEvent.sourcePort);
            receivedMessage.Type=REQ_UNKNOWN;
            break;

      default:
            receivedMessage.Type=REQ_UNKNOWN;
            break;
      }
}

UnsubscribeVariables();UnsubscribeEvent();//disconnecting initially connected ports
//resetting module
return 0;
}
```

Figure 8: Structure du comportement d'un module

Dans les réactions associées aux types des messages reçus, on peut distinguer deux catégories:

- Les types de messages pour lesquels les différentes spécialisations possibles de ce squelette de module pourront fournir une implémentation propre. Dans cette catégorie on retrouve les réactions aux requêtes d'activation (`REQ_START`), d'arrêt (`REQ_STOP`), d'acquittement (`REQ_ACK`), de terminaison (`REQ_KILL`), de configuration (`REQ_SETPARAM`) et de notification (`REQ_EVENT`).

- Les types de messages pour lesquels la réaction est fixée et non modifiable par les spécialisations du squelette de modules. Dans cette catégorie on retrouve toutes les réactions aux requêtes liées à la gestion de la connexion des ports de données (`REQ_SUBSCRIBE` et `REQ_UNSUBSCRIBE`) et d'événements (`REQ_SUB_EVENT` et `REQ_UNSUB_EVEND`). Pour chacun de ces types de message la réaction consiste à **immédiatement** appeler la primitive adéquate de *libmodule* pour gérer l'abonnement ou le désabonnement à un port producteur (de données ou d'événement) du module en utilisant pour arguments les paramètres contenus dans la requête (cf. fig. 8). Ainsi la gestion du « bon moment » pour la connexion des modules est reportée intégralement sur les modules chargé d'établir les connexions, en l'occurrence les superviseurs. Ceci loin d'être une contrainte est une sécurité car les superviseurs sont les seuls, chacun à leur niveau, à même de connaître les états d'exécution de l'ensemble des modules concernés et donc à même de réaliser des séquences de connexion valides au regard de ces états.

Nous avons à partir de ce squelette générique définit différentes spécialisation afin que les différents types de modules présents dans une architecture soient conformes à la spécification de notre décomposition logicielle et aux algorithmes utilisés dans les modules génériques. Notons que dans les explication ci-après, le fait qu'un module ne réagisse pas à certains types de messages se traduit simplement dans le code par le fait que le bloc `case` correspondant à ce type de message dans la boucle d'attente active n'est pas présent.

Le **module ordonnanceur** ne réagit pas aux requêtes **d'activation** et **d'arrêt**, son activité étant déclenchée par la réception de requêtes de **configuration** qui correspondent à des demandes d'ordonnancement ou d'abandon de **schémas périodiques** (cf. section 2.1.3), provenant d'un ou plusieurs modules superviseurs. Il ne réagit pas non plus à des requêtes **d'abonnement** ou de **désabonnement** à ses ports de sortie de **données** puisqu'il ne possède pas de tel ports. Il réagit également aux requêtes **d'acquittement** d'exécution (exécution finie ou interrompue) en provenance des modules synchrones qu'il ordonnance. Enfin, il ne réagit à aucune requête de **notification** d'événements puisqu'il ne peut s'abonner à aucun événement.

Les **modules superviseurs** ne réagissent pas aux requêtes **d'activation** et **d'arrêt**, leur activité étant déclenchée par la réception de requêtes de **configuration** qui correspondent à des demandes d'exécution ou d'abandon de **fonctions de supervision** (cf. section 2.1.3) provenant d'un ou plusieurs modules superviseurs de niveau hiérarchique supérieur. Il ne réagissent pas non plus à des requêtes **d'abonnement** ou de **désabonnement** à leurs ports de sortie de **données** puisqu'ils ne possèdent pas de tel ports. Il réagit également aux requêtes **d'acquittement** d'exécution (exécution démarrée ou interrompue) en provenance des modules asynchrones dont il contrôle l'activité

(modules d'interaction ou module temps-restant). Enfin, ils réagissent aux requêtes de **notification** provenant des modules de la couche exécutive ou des superviseurs de niveau hiérarchique inférieur auxquels ils se sont eux-mêmes abonnés.

Les **modules asynchrones** (d'interaction ou temps-restant) réagissent à tous les types de requêtes excepté les demandes **d'abonnement** ou de **désabonnement** à leurs ports de sortie de **données**, puisqu'ils n'en possèdent pas, et les requêtes **d'acquittement,** puisqu'ils ne doivent pas envoyer des requêtes d'exécution directement d'autres modules (les modules asynchrones ne peuvent communiquer entre eux et en direction des superviseurs qu'à travers des envois d'événements).

Les **modules synchrones** réagissent à tous les types de requêtes excepté les requêtes **d'acquittement,** puisqu'ils ne doivent pas envoyer des requêtes d'exécution directement à d'autres modules (ils ne communiquent entre eux qu'à travers des flux de données et en direction des superviseurs que par envoi d'événements) et les **requêtes de notification**, puisqu'ils ne possèdent pas de port d'entrée d'événements. Notons également que leur réaction à des requêtes de **configuration** est **limitée à la seule mise à jour de leur paramètres internes** et **ne peuvent engendrer d'autres réactions** (contrairement à ce qui peut être fait dans les modules asynchrones).Ceci se justifie par le fait que l'activité (notamment sa durée) d'un module synchrone doit être totalement contrôlable par l'ordonnanceur qui exploite les requêtes d'activation et d'arrêt à cette fin. Il ne faut donc pas que la fixation d'un paramètre du module par un superviseur puisse remettre en cause cette propriété. La mise à jour d'un paramètre se faisant en temps constant et faible comparativement aux calculs que réalise le module synchrone, c'est la seule opération qui a été autorisée.

L'implémentation du squelette de code spécialisé pour les modules synchrones et asynchrones est similaire pour ce qui concerne la gestion des requêtes d'activation et d'arrêt. Il s'agit de gérer:

- Le lancement de l'activité nominale du module. L'activité nominale d'un module synchrone ou asynchrone est implémentée dans une **fonction** définie par l'utilisateur (code spécifique à l'application).
- L'interruption de l'activité nominale du module. Il s'agit de permettre d'arrêter l'exécution de la fonction implémentant l'activité nominale puis d'envoyer automatiquement un acquittement d'exécution vers le bon destinataire. Pour cela, la fonction doit en quelque sorte être interruptible ce sans quoi le module ne pourra réagir à la requête avant la fin de son exécution. Pour cela, la fonction doit décomposer les calculs en plusieurs étapes (à la discrétion de l'utilisateur), le module regardant entre chaque étape si une requête n'a pas été reçue (en utilisant la primitive non bloquante `LookForRequest`). Si une requête a été reçue le module y réagit puis reprend l'exécution de la fonction à l'étape suivante sauf si la réaction implique une interruption de l'exécution (i.e. réaction à des requêtes **d'arrêt** ou de **terminaison**).
- Pour les modules synchrones uniquement, il émettent une **requête d'acquittement** lorsque la fonction implémentant l'activité nominale a franchi toutes les étapes d'exécution.

Les **modules synchrones et asynchrones** étant les briques de bases très spécifiques à partir desquelles sont construites les architectures de contrôle robotique, une partie de leur code comportemental doit être écrit par l'utilisateur. Ils déclarent trois fonctions qui demandent à être

implémentées : la fonction d'initialisation du module, la fonction de terminaison du module et la fonction implémentant leur activité nominale. Ces trois fonctions sont automatiquement appelées par leur squelettes de code.

Les **modules ordonnanceur et superviseurs** sont quant à eux des modules génériques dont les mécanismes internes sont totalement prédéfinis dans des fonctions auxiliaires qui sont directement appelées dans leur squelette de code. Le seul rôle de l'utilisateur est de définir les **fonctions d'initialisation des structures de données** exploitées par ces mécanismes.

2.2.3 Mécanismes des modules génériques

Le middleware fournit une implémentation des mécanismes génériques utilisés au sein d'une architecture de contrôle ContrACT. Les deux sous-section suivantes présentent donc dans l'ordre le mécanisme d'ordonnancement et le mécanisme de supervision. Comme nous l'avons stipulé dans la section précédente les modules correspondant (ordonnanceur et superviseurs) sont en grande partie génériques : seule leurs configurations changent suivant l'application (i.e. robot et type de mission qu'il est capable de réaliser) mais pas les algorithmes qui exploitent ces configurations. Nous présentons donc par la suite les types de données et le fonctionnement des mécanismes qu'ils intègrent.

2.2.3.1 Ordonnancement

Le mécanisme d'ordonnancement est le cœur des architectures développées récemment au LIRMM ***[Eljalaoui 2007]*****,** notamment de l'architecture des robot sous-marin TAIPAN. L'idée d'intégrer un ordonnanceur applicatif provient d'une part de la volonté d'avoir un système temps-réel observable et contrôlable et d'autre part de la nécessité de contrôler l'utilisation de l'instrumentation embarquée (facteurs très important en robotique sous-marine pour limiter les problèmes d'interférence entre sonar).

La figure 9, reprise de ***[Eljalaoui 2007]***, illustre l'utilisation et le fonctionnement global du mécanisme d'ordonnancement. Le module ordonnanceur possède deux « bases de données » (qui sont dans la version actuelle des zones en mémoire vive):

- Une répertoriant pour chaque module l'ensemble de ses propriétés d'ordonnancement (type de tâche et durée) et répertoriant les relation de compatibilité entre modules (deux modules sont incompatibles -i.e. ne peuvent être activé en même temps - si par exemple ils interagissent avec des dispositif interférents).

- Une répertoriant pour chaque **schéma périodique** connu par l'ordonnanceur, les informations d'ordonnancement de ce schéma: les modules impliqués dans le schéma ainsi que le graphe de précédence entre ces modules, et les deux paramètres temporels utilisé par l'algorithme d'ordonnancement: la **période** (de répétition du schéma) et le **délai critique** (délai maximal de temps écoulé pour l'exécution de l'ensemble des modules du schéma dans une période).

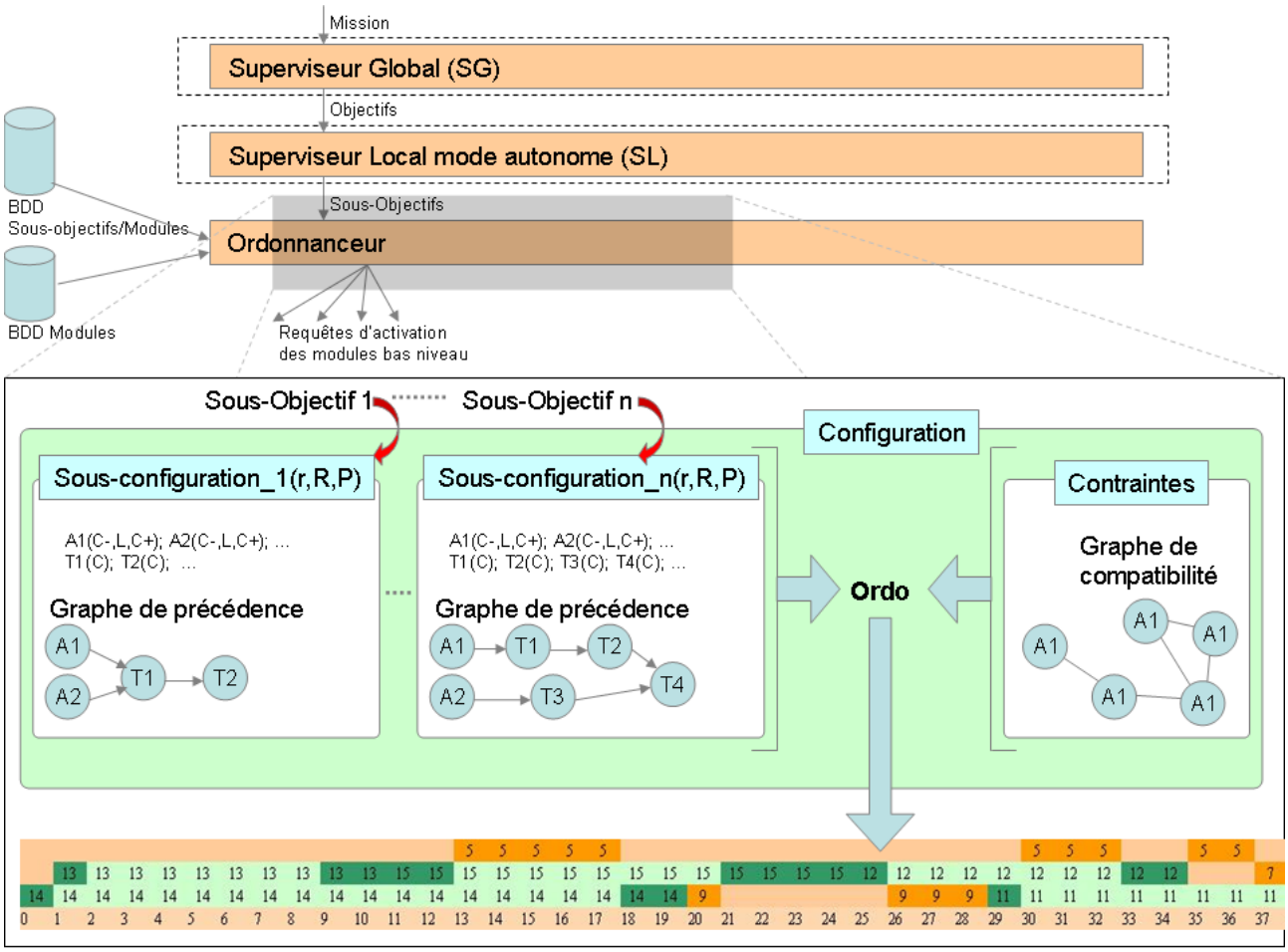

Figure 9: représentation schématique du mécanisme d'ordonnancement dans l'architecture

L'ordonnanceur reçoit des sous-objectifs des superviseurs qui correspondent à des demandes d'ordonnancement (ou d'abandon) de **schémas périodiques** du point de vue du module ordonnanceur (cf. figure 9). Le mécanisme d'ordonnancement utilise les informations relatives au schéma et aux modules utilisés pour créer une **sous-configuration** qui est une structure de donnée intermédiaire contenant toutes les données nécessaires à l'ordonnancement du schéma. Il intègre par la suite cette sous-configuration à la **configuration** active (qui contient les sous-configuration déjà ordonnancées) afin d'avoir une donnée globale traduisant le problème d'ordonnancement. Le calcul d'une sous-configuration revient à déterminer les délais critiques de tous les modules qu'elle contient, à partir des informations temporelles et du graphe de précédence.

A partir de cette configuration il lance effectivement le calcul de l'ordonnancement qui consiste à définir l'échéancier à exécuter. Notons que l'échéancier est à horizon court c'est-à-dire qu'il intègre uniquement les modules synchrones **en cour de traitement** à partir du prochain pas de temps. Il peut y avoir plusieurs modules synchrones en cours de traitement à un pas de temps donné car il existe des modules qui peuvent spécifier une **période de latence** pendant laquelle ils sont inactifs car en attente d'une entrée/sortie. Ces modules sont ordonnancés comme un type de tâches particulier (du point de vue de l'ordonnanceur applicatif) appelé **tâche d'interaction** par opposition aux **tâches de traitement** qui n'ont pas de période de latence. Les tâches d'interaction sont utilisées

pour intégrer des modules interagissant de manière potentiellement longue avec des entrées/sortie ou avec des dispositifs interférents (e.g. sonars interférents, communication sur réseau sans fil interférant, etc.). A un pas donné de l'échéancier, il ne peut donc y avoir un ensemble de modules ordonnancés comme des tâches d'interaction en période de latence mais un seul module ordonnancé comme une tâche de traitement.

Le calcul de l'échéancier est donc relativement simple mais surtout très réactif : il consiste principalement à définir quel sera le prochain module à exécuter suivant une stratégie *earliest deadline first* , basée sur les délais critiques préalablement calculés. Le calcul de l'échéancier suit l'algorithme représenté schématiquement dans la figure 10 :

- les **tâches éligibles** sont les modules pour lesquels tous les modules précédents dans le graphe de précédence ont terminé leur exécution.
- La **priorité** d'un module est calculée en fonction de sa date d'activation, de son délai critique et de son type de tâche (interaction ou traitement, les tâches d'interaction étant plus prioritaire et pouvant préempter les tâches de traitement).

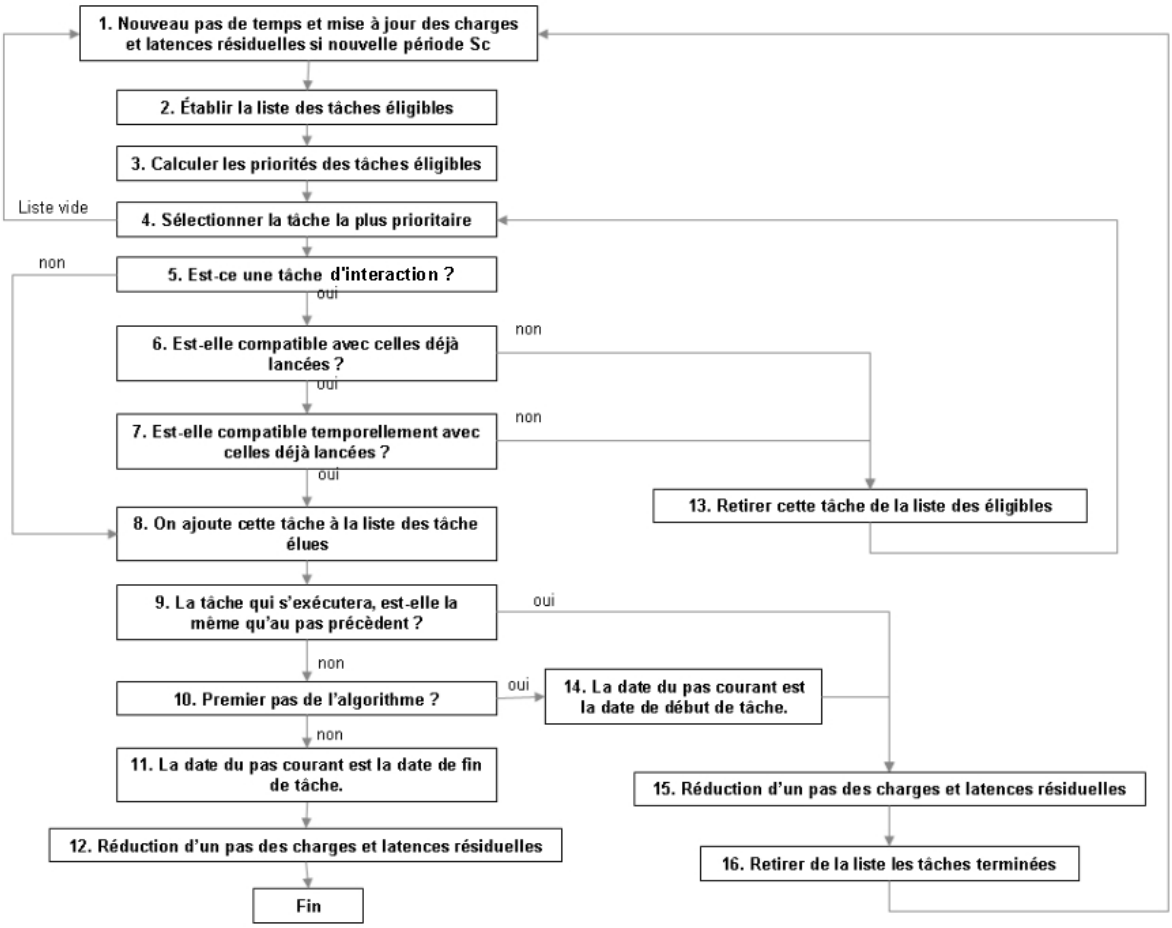

Figure 10: Calcul de l'échéancier de l'ordonnanceur, extrait de *[Eljalaoui 2007]*

- Pour les modules ordonnancés comme des tâches d'interaction, leur **compatibilité** s'établit en fonction du graphe de compatibilité (étape 6, fig. 10).
- Pour les modules ordonnancés comme des tâches d'interaction, leur **compatibilité temporelle** (étape 7, fig. 10) s'établit en fonction leur périodes de latence et d'exécution : un tel module ne sera choisi que s'il peut s'exécuter (période avant ou après la période de latence) durant le **temps de latence** laissé par les modules déjà en période de latence.

De façon générale, le module ordonnanceur se réveille périodiquement par l'utilisation d'un timer qui émet une requête de configuration particulière (réveil) dans son propre port de requête et il effectue l'action courante définie dans son échéancier. Si un module n'a pas produit d'acquittement dans le temps d'exécution qui lui est affecté lorsque l'échéancier stipule que le module doit être arrêté, l'ordonnanceur lui laisse du temps supplémentaire pour pouvoir s'exécuter ou l'arrête si le module est en retard plusieurs fois. Si le module commet trop de fautes temporelles sur un horizon de temps donné, l'ordonnanceur notifiera les superviseurs qu'une faute temporelle critique a été rencontrée. De manière équivalente si le calcul de l'échéancier révèle trop de faute temporelles d'un schéma périodique (tous les modules du schéma n'ont pu être exécutés sur la période requise) sur un horizon de temps (i.e. rapport nombre de fautes temporelles sur nombre de période) donné l'ordonnanceur notifiera également les superviseurs qu'une faute temporelle critique a été rencontrée

2.2.3.2 Supervision

Le mécanisme de supervision a été défini dans le but de produire de façon simple des superviseurs capable d'exploiter les possibilité offertes par l'ordonnanceur et plus généralement par le modèle de programmation par modules. L'idée générale du mécanisme a été pour la première fois développé dans ***[Eljalaoui 2007]*** et elle visait originellement a fournir des moyens pour intégrer et exploiter facilement des modules dans l'architecture.

Néanmoins les projets dans lesquels le LIRMM est impliqué ont fait émerger des besoins nouveaux, dont la satisfaction a nécessité de revoir complètement le mécanisme de supervision afin de le rendre beaucoup plus souple et puissant:

- possibilité d'utiliser des clauses d'événements dans les conditions et de définir des temps de persistance aux événements pendant lesquels leur occurrence reste valide.
- possibilité de manipuler des variables pour stocker et exploiter les données associées aux événements, afin de permettre par exemple de configurer des schémas périodiques ou des modules à partir de ces données (e.g. permettre à une loi de commande d'être paramétrée avec une consigne désignant le point à atteindre obtenue par un calcul de planification de chemin)
- possibilité de valider l'occurrence d'un événement en effectuant des tests sur sa donnée associée.
- possibilité de gérer des schémas événementiels capable afin de gérer les interactions événementielles entre le contrôleur et certains système externes
- possibilité de lancer des calculs en temps restant, via l'utilisation de module asynchrones.
- possibilité d'activer (et pas seulement de désactiver) des règles à partir d'événements produits par des modules.
- possibilité de créer des fonctions de supervision ayant un comportement cyclique, etc.
- possibilité de créer des hiérarchiques complexes de superviseurs afin de pouvoir raffiner aussi précisément que souhaité les modules de fonctionnement.

Afin de comprendre l'algorithme de supervision, nous expliquons de manière abstraite la forme que prend une **fonction de supervision**. Une fonction de supervision possède un ensemble de **paramètres** formels typés identifiés de manière unique par un nom, un ensemble de **variables locales** typées identifiées de manière unique par un nom, et un ensemble de **règles** identifiées de manière unique par un label, soit schématiquement:

*fonction(TypeConstruit1 paramètre1, TypeConstruit2 paramètre2, **...**){*
***variables locales**{*
TypeConstruit1 a = paramètre1;
TypeConstruit2 b;
int c = parametre2.x;
...
}
***règles**{*
X: [disjonction de clauses d'événement] ***séquence_actions*** *[disjonction de clauses d'événement]*
...
}
}

Les paramètres formels servent uniquement à initialiser les variables locales grâce à des affectations. Un superviseur contient pour chaque fonction la liste de ses paramètres potentiels ainsi que les informations de type à partir desquelles il pourra dé-sérialiser (lors de la réception d'un appel de fonction de supervision) et sérialiser (lors de l'émission d'un appel d'une fonction de supervision d'un autre superviseur) les paramètres d'appels. Il possède également, pour chaque fonction de supervision les références vers les variables représentant ses variables locales et ses paramètres ainsi que les procédures d'initialisation des variables locales soit à partir des paramètres reçus, soit par défaut. Enfin à chaque fonction de supervision est associée une liste de règles, chaque règle possédant un **label**, une **pré-condition** décrite par une disjonction de clauses d'événements, une **postcondition** décrite par une disjonction de clauses d'événements, et une séquence d'**actions**. Le superviseur possède une **table de ses objectifs** qui représente le point d'accès à l'ensemble de ses **fonctions de supervision**.

Si le principe des fonctions de supervision construites à partir de règles est resté le même que dans ***[Eljalaoui 2007]***, la réalisation de ce principe, tant au niveau de la définition des fonctions de supervision que du mécanisme chargé de les exécuter a complètement changé. Son fonctionnement global est représenté schématiquement dans la figure 11.

Avant de rentrer dans les explications du fonctionnement, il faut préciser qu'il existe trois types d'événements gérés par le mécanisme de supervision:

- les **événements de module** correspondent aux événements produits par les modules de la couche exécutive ou les modules superviseurs hiérarchiquement inférieur au superviseur considéré, auxquels le superviseur est **abonné**. Un cas spécifique sont les événements produits par le module ordonnanceur, auxquels tous les superviseurs sont automatiquement abonnés.

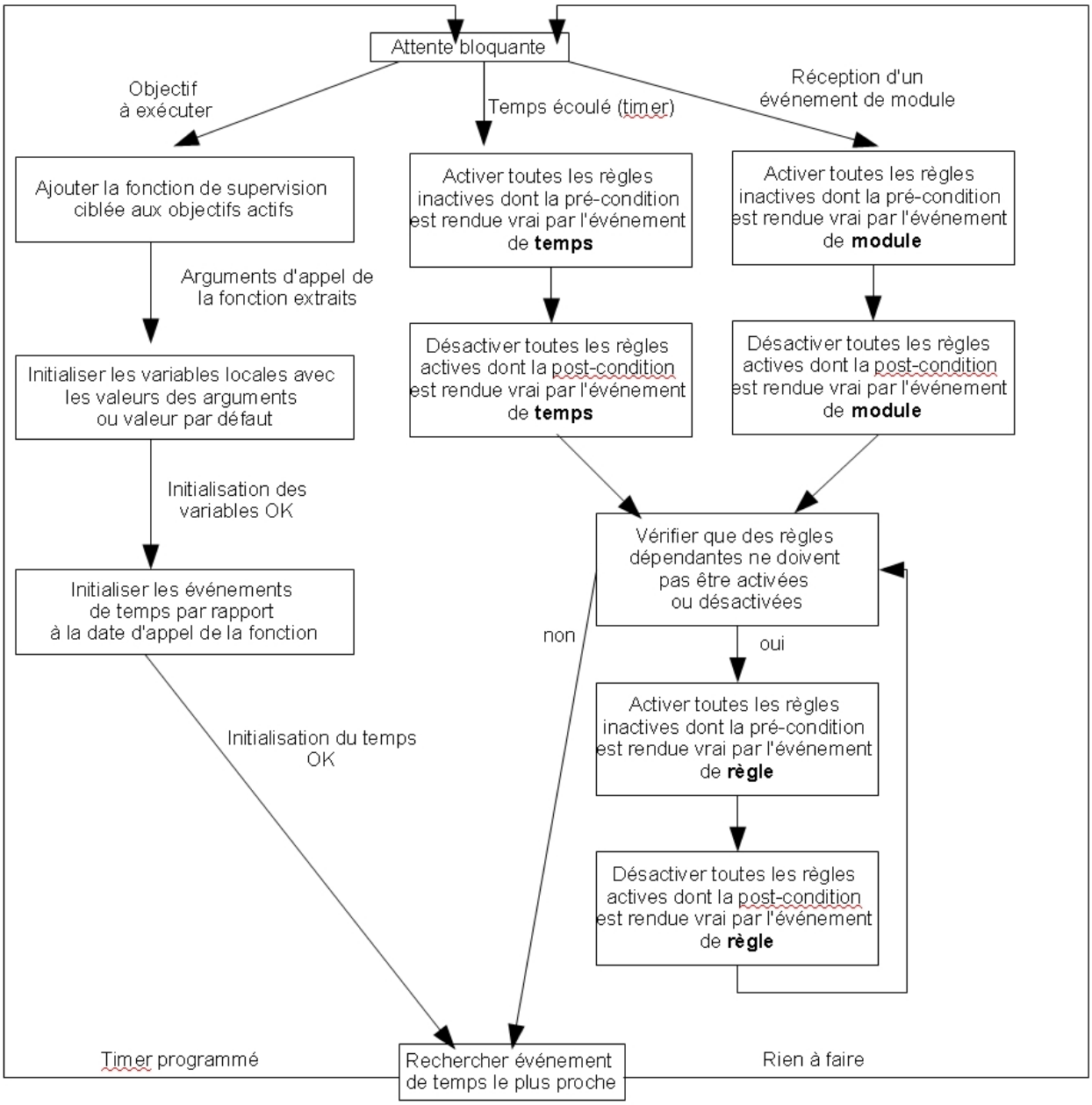

Figure 11: représentation schématique du comportement d'un superviseur

- Les **événements de règle** correspondent à des événements internes au superviseur qui apparaissent dès lors qu'une règle devient active ou inactive et que cette activation ou cette désactivation entraine l'activation ou la désactivation d'autres règles.
- Les **événements de temps** correspondent aux événements générés par la programmation, par le mécanisme de supervision lui-même, d'un **timer apériodique** qui va envoyer une requête de configuration dans le ports de requête du superviseur et une donnée de temps dans un de ses ports de paramètre prévu à cet effet. Les événements de temps sont de deux types : soit **absolu** (ils correspondent à l'écoulement d'une quantité de temps depuis l'activation de la fonction), soit de **durée d'exécution d'une règle** (date à laquelle une règle s'est exécutée pendant un temps donné).

Le superviseur est initialement en attente de requêtes de réalisation d'objectifs. Dès lors qu'il reçoit une requête (*Objectif à exécuter*, fig. 11) il ajoute la fonction de supervision ciblée par la requête aux objectifs actifs de sa table des objectifs. Il extrait ensuite les éventuels arguments associés à la requête et initialise les variables locales de la fonction de supervision, comme expliqué précédemment. Il initialise ensuite les **événements de temps absolu** (i.e. ceux relatifs à la date d'appel de la fonction) si il y en a qui sont spécifiés dans les conditions des règles de la fonction. Il recherche ensuite l'événement de temps le plus proche qui va le réveiller, programme son timer sur la date adéquate, et se met à nouveau en attente bloquante. Les **événements de temps absolu** vont ainsi servir à déclencher le démarrage de la première règle de la fonction qui sera activée, qui peuvent être vue comme un point d'entrée de l'exécution de la fonction. Notons que le mécanisme d'**abandon d'un objectif** suit exactement la même logique que pour l'exécution d'un objectif.

Le superviseur peut être réveillé par une notification de son timer (**événement de temps**), ce qui sera le cas par exemple pour la ou les premières règles à activer après réception d'un appel de fonction de supervision. Dans ce cas le mécanisme de supervision va activer ces règles, et durant les exécutions suivantes il pourra également désactiver des règles actives si celles-ci contiennent des condition vérifiées par les **événements de temps**. Puis le mécanisme de supervision vérifie si des règles dépendantes de ces règles nouvellement activées doivent également être activées ou désactivées. Enfin il se remet en état d'attente. Lors des exécutions suivantes, les règles activées peuvent elle-même être à l'origine de la programmation du timer (événement relatif à une durée d'exécution de règle atteinte), dans ce cas le superviseur est réveillé de la même manière et effectue la même séquence d'opérations.

La troisième possibilité de réveil du superviseur est liée à la réception d'un **événement de module**. Dans ce cas le superviseur va activer toutes les règles inactives pour lesquelles l'occurrence de l'événement rend leur précondition vraie et va désactiver toutes les règles actives pour lesquelles l'occurrence de l'événement rend leur postcondition vraie. De la même manière que pour événements de temps, il va ensuite vérifier les dépendances entre règles, c'est-à-dire l'occurrence d'**événements de règles**.

Grâce à ce mécanisme, l'ensemble des règles d'une fonction sont évaluées dès que, et seulement quand, nécessaire, ce qui permet d'une part de s'assurer de l'aspect réactif de l'exécution des superviseurs et d'autre part de créer des comportements plus ou moins complexes, ayant un fonctionnement séquentiel (enchainement simple de règles), parallèle (exécution de plusieurs règles en parallèles), cyclique (réactivation cyclique des règles), conditionnel (activation conditionnelle de règles) ou n'importe quelle combinaison de tous ces fonctionnements.

Pour comprendre comment la définition de ces comportements est possible il faut aborder la problématique de l'évaluation d'une règle. Comme décrit auparavant, chaque règle des objectifs actifs est évaluée dès qu'un **événement significatif** doit être géré par le mécanisme de supervision. Si une règle est active, alors sa **postcondition est évaluée** et si elle est vrai la règle est désactivée, c'est-à-dire que l'**exécution de sa séquence d'actions se termine**. A l'inverse si elle est inactive c'est **sa précondition qui est évaluée** et si elle est vrai la règle est activée, c'est-à-dire que l'**exécution de sa séquence d'actions commence**. Les mécanismes cruciaux pour comprendre l'évaluation d'une règle sont donc l'**évaluation des conditions** et l'**exécution d'une séquence d'action**.

L'évaluation des (pré et post) conditions se fait **séquentiellement** sur chaque la liste des clauses. Dès qu'une clause est vrai, la condition devient vrai sans évaluation des autres clauses. L'évaluation d'une clause se fait par évaluation de **tous ses événements**. Si tous les événements sont valides alors la clause est vrai. Tous les événements d'une clause se voient attribué un **temps de persistance** pendant lequel **l'événement reste valide après occurrence**. Pour qu'un événement soit vrai il faut donc soit qu'il vienne d'être produit, soit que son temps de persistance ne soit pas dépassé. Par ailleurs, les **événements de modules** peuvent voir leur **validité conditionnée** : dès leur occurrence un ensemble de tests est effectuée sur la donnée associée à l'événement, et si tous ces test sont vrais alors l'événement est valide, sinon l'événement est **considéré comme non reçu pour la clause considérée uniquement**.

L'exécution d'une séquence d'actions revient à exécuter dans l'ordre spécifié la séquence des **actions primitives**. S'il s'agit d'une activation de règle, chaque action est exécutée, s'il s'agit d'une désactivation de règle chaque action est annulée (si possible pour l'action considérée). Les actions primitives possibles sont :

- Lancement (activation de règle) ou annulation (désactivation de règle) d'un **schéma périodique**. Cette action revient pour le superviseur à abonner (respectivement désabonner pour la désactivation) les flux de données des modules synchrones contenus dans le schéma, de paramétrer ces modules si besoin est, puis d'envoyer une requête d'ordonnancement (respectivement d'abandon) au module ordonnanceur.
- Lancement (activation de règle) ou annulation (désactivation de règle) d'un **schéma événementiel**. Cette action revient pour le superviseur à abonner (respectivement désabonner pour la désactivation de règle) les flux d'événements des modules asynchrones contenus dans le schéma, de paramétrer ces modules si besoin est, puis d'activer (respectivement arrêter) un à un ces modules.
- **Paramétrage en ligne** d'un module synchrone ou d'un module asynchrone (ne peut être annulé dans le cas de la désactivation de règle).
- Lancement (activation de règle) ou annulation (désactivation de règle) d'une **fonction de supervision d'un superviseur hiérarchiquement inférieur**. Cette action revient pour le superviseur à sérialiser les arguments d'appels de la fonction puis d'émettre une requête d'exécution (respectivement d'abandon pour la désactivation de règle) d'objectif.
- Lancement (activation de règle) ou annulation (désactivation de règle) d'un **calcul en temps restant**. Cette action consiste pour le superviseur à donner une priorité faible au module asynchrone implémentant le calcul, à le paramétrer puis à lancer son exécution (respectivement arrêter son exécution pour une désactivation de règle).
- Exécution d'un **calcul critique**. Cette action revient pour le superviseur a effectuer un appel de fonction local prenant en paramètres certaines de ses **variables locales**. Cette action ne peut être annulée dans le cas de la désactivation de règle puis qu'étant critique elle bloque le fonctionnement du superviseur jusqu'à ce quelle se termine.
- Abonnement (activation de règle) ou désabonnement (désactivation de règle) à un **flux de données d'événements**. Cette action consiste pour le superviseur à établir une connexion (respectivement à détruire une connexion pour la désactivation de règle) entre un de ses ports d'entrée d'événements et un port de sortie d'événement d'un module de la couche exécutive ou d'un module superviseur hiérarchiquement inférieur.
- **Notification d'un événement**. Cette action consiste pour le superviseur à notifier via un de ses ports de sortie d'événements, l'occurrence d'un événement qu'il a détecté. Cette action ne peut être annulée dans le cas d'une désactivation de règle car la notification est instantanée et unique dès lors que la règle devient active.

Les **variables locales** sont essentielles afin de permettre un haut degré d'adaptation des règles aux valeurs réelles représentant l'état du robot ou de son environnement. Elle peuvent être utilisées à différents endroits des fonctions de supervision:

- dans les **actions primitives**, à des fins, par exemple, de paramétrage des modules ou pour mettre à jour la valeur associé à un événement notifié par le superviseur.
- dans les **pré et post conditions**, au moment de l'évaluation de la validité d'un événement de module (cf. plus haut).
- dans les **pré et post conditions**, au moment de l'occurrence d'un événement dont la condition a été respecté, ou leurs valeurs peuvent être modifiées par la donnée associée à l'événement (extraction de la donnée d'événement en vue de sa mémorisation partielle ou complète).

2.2.4) Conclusion sur le middleware

Le middleware offre un ensemble de moyen très variés et des mécanismes génériques pour implémenter une architecture de contrôle. Néanmoins son utilisation peut s'avérer complexe si elle n'est pas guidée et partiellement automatisée. En effet il faut par exemple construire de façon adéquate les différentes structures de données exploitées par les mécanismes génériques (superviseur, ordonnanceur), ce qui peut s'avérer être une tâche pénible et génératrice d'erreurs. D'autre part il requiert une certaine concentration dans l'utilisation des différents types de modules pour s'assurer notamment:

- que le bon squelette de module est utilisé à bon escient (i.e. ne pas utiliser un squelette de module synchrone pour un module asynchrone).
- qu'un seul module ordonnanceur est bien présent.
- que les types de données échangés entre modules (via leurs ports) sont bien compatibles.
- que les modules respectent les contraintes systèmes (taille des noms de module notamment, priorités affectés aux différents types de modules)
- etc.

Afin de limiter drastiquement le nombre d'erreurs, nous avons développé un outils de développement logiciel chargé de s'assurer, autant que possible, que les développeurs d'architecture de contrôle logicielles respectent la méthodologie globale et utilisent correctement le middleware pour ce faire.

2.3) Outil logiciel support au développement

L'outil logiciel développé pour supporter la méthodologie de développement a été réalisé comme une collection de plugins pour l'environnement ECLIPSE. L'idée générale a été de créer un **langage dédié** (domain specific language, DSL) au développement d'architectures de contrôle logicielles suivant la méthodologie présentée en section 2.1. Ce langage est utilisé pour décrire les interfaces des modules, ainsi que la configuration des modules génériques (ordonnanceur et superviseur). L'outil permet de réaliser des vérifications syntaxiques et sémantique sur les modèles générés en internes à partir des descriptions faites dans ce langage, mais également de **générer le code source directement interfacé avec le middleware** développé et présenté en section 2.2.

Pour la réalisation de cet environnement de développement, plusieurs technologies ECLIPSE pour le développement orienté modèle (model driven engineering, MDE) ont été utilisées:

- **EMF** : c'est le cœur des technologies MDE dans ECLIPSE, qui fournit toutes les primitives nécessaires à la manipulation de modèles et de méta-modèles.
- **Xtext** : C'est le langage utilisé pour décrire la grammaire du langage dédié et générer à la fois les parseurs et les éditeurs de code et de modèles EMF pour ce langage.
- **Check/Xtend** : c'est le langage utilisé pour implémenter les vérifications sémantiques du langage dédié.
- **Xpand/Xtend** : c'est la langage de templates utilisé pour la génération de code source.

2.3.1) Organisation d'un projet ContrACT

L'environnement de développement ContrACT propose de décomposer les projets d'architecture de contrôle en deux parties. Une partie contenant le code source, c'est-à-dire les fichiers écrits dans le langage dédié, et l'autre partie correspondant au code généré, c'est-à-dire des fichiers source et d'entête C, une arborescence de répertoires et des fichiers de compilation (makefile), comme présenté dans la figure 12.

Tous les fichiers sources écrit dans le langage dédié **Cactal** (pour ContrACT Advanced Language) possèdent la même extension (*.contract*) mais sont différenciés en fonction de leur contenu:

- *les fichiers de définition de type*. Ces fichiers contiennent la définition des types de données utilisés dans les échanges entre les modules utilisés dans l'architecture. L'existence des types de données échangés au niveau du langage Cactal se justifie par une volonté de rendre le développement plus fiable en faisant du test de type lors de la connexion des modules: on peut ainsi savoir si des modules échange des données de même type et si ce n'est pas le cas reporter une erreur. A partir de ces types de données, les structures C équivalente et les différentes fonctions servant à leur manipulation dans le middleware sont automatiquement générées. Les fichiers de définition de type peuvent également contenir des déclarations de fonctions que l'utilisateur devra implémenter en C. Nous verrons par la suite comment utiliser ces fonctions.

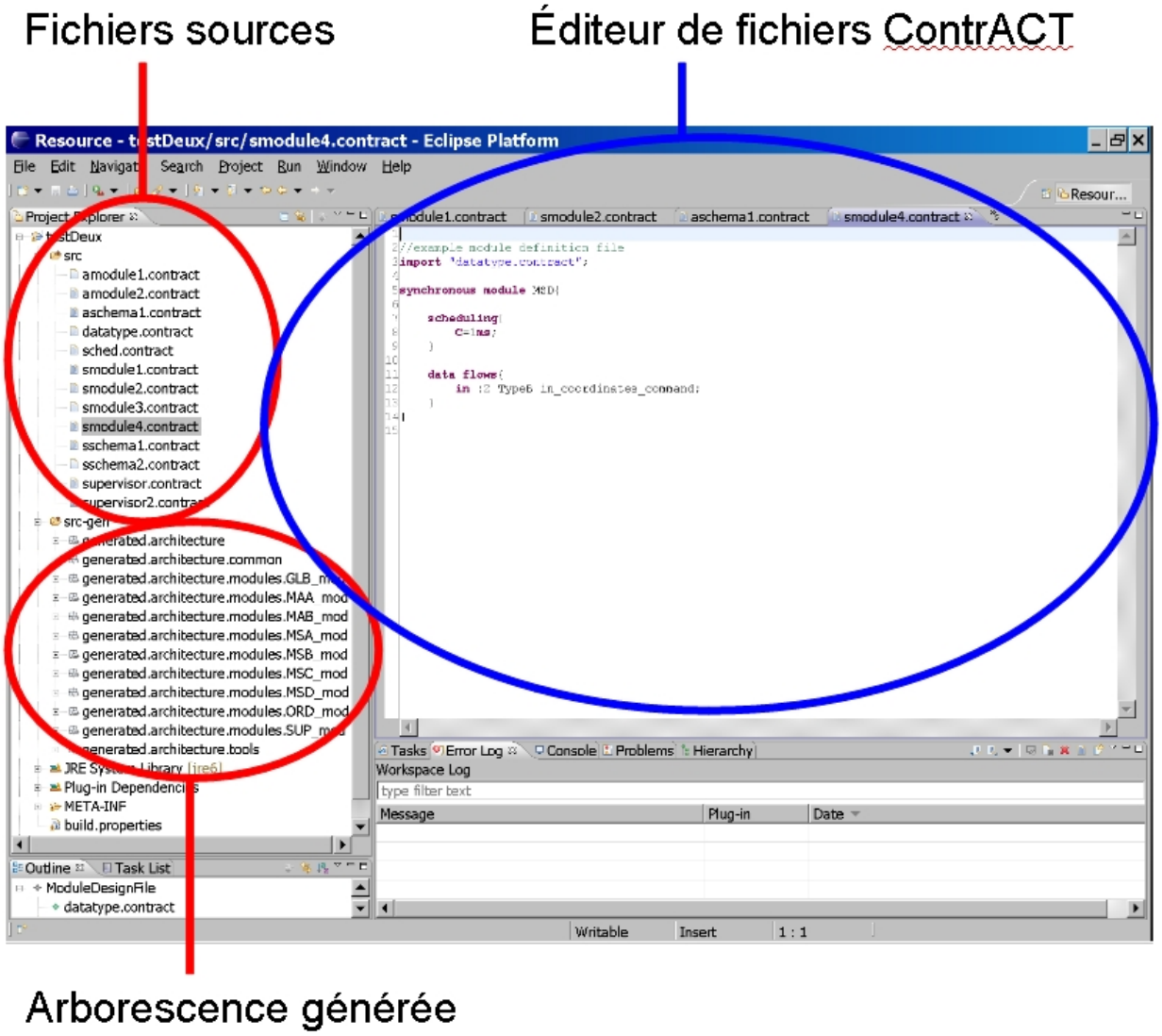

Figure 12: Vue globale sur l'organisation d'un projet dans l'outil de développement

- *Les fichiers de définition d'interface de module fonctionnel*. Ces fichiers servent à définir les interfaces et propriétés essentielles des modules dit « fonctionnels », c'est-à-dire les modules contenant les fonctions de base programmées (en C) par l'utilisateur. Chaque fichier correspond donc à l'interface d'un module **synchrone** ou **asynchrone (temps-restant ou d'interaction)**. Une interface contient le nom du module, la définition de ses ports de paramètres, d'entrée / sortie de données (module synchrone uniquement) ou d'événements (module asynchrone uniquement pour les ports d'entrée d'événements) de façon conforme à la méthodologie. Les modules synchrones possèdent en plus une spécification de leur paramètre de **temps d'exécution nominal,** utile à l'ordonnanceur. A partir de ces informations des squelettes de code complets des modules fonctionnels sont générés et l'utilisateur n'a plus qu'à remplir les fonctions implémentant leur comportement (initialisation, terminaison, exécution nominale, réaction arrivée de paramètres, réaction aux événements).

- *Les fichiers de définition des schémas*. Ces fichiers servent à définir les **schémas périodiques ou événementiels** utilisés dans l'architecture, c'est-à-dire les assemblages de modules (via leur ports de données pour les schéma périodiques ou d'événement pour les schéma événementiels) et leurs paramétrage, ainsi que, pour les schémas périodiques les propriétés nécessaires à leur ordonnancement (période, délai critique et graphe de précédence). Chaque fichier correspond à un schéma, mais ne génèrera pas de module particulier: il sert à générer du code à plusieurs endroits dans l'architecture logicielle, dans l'ordonnanceur (pour les schémas périodiques) et dans les superviseurs utilisant ces schémas.

- *Le fichier de définition de la configuration de l'ordonnanceur*. Ce fichier unique permet de répertorier l'ensemble des schémas périodiques et des modules synchrones connus par l'ordonnanceur. Il sert à la globalisation des informations nécessaires à la génération du module ordonnanceur.

- *Les fichiers de définition des superviseurs*. Chaque fichier correspondant à la définition d'un **module superviseur** soit : son interface, constituée de ses ports d'événement en entrée et en sortie ; son comportement constitué d'un ensemble de fonctions de supervision définissant l'ensemble des objectifs qu'il est capable de réaliser. L'utilisation du langage *Cactal* pour décrire les superviseurs est détaillée dans la sous-section suivante. La génération de code utilisera les informations contenues dans un tel fichier pour **générer le code complet des modules superviseurs**.

2.3.2) Description des superviseurs

La description des superviseurs est l'apport majeur de l'environnement de développement du langage *Cactal*. Cette description est complète dans le sens ou l'interface mais aussi le comportement des superviseurs sont complètement définis, l'utilisateur n'ayant pas à personnaliser le code des superviseurs.

2.3.2.1) Interface

L'interface d'un superviseur est simplement décrite par une ensemble de **ports d'événements**, et par la **déclaration d'un ensemble de fonctions de supervision**., de la manière présentée dans la figure 13.

```
supervisor SUP{

event flows{
in :1 TypeC message_received;
in :2 string connection_status;
out :1 string failure_message;
}

fonction1(TypeA consigne){
...
}

fonction2(TypeB param){
...
}
```

Figure 13: Exemple d'interface d'un superviseur

L'ensemble des ports d'événements en entrée servent à récupérer les événements auxquels le module superviseur est abonné et qui sont produits dans la couche exécutive ou par les superviseurs de niveau hiérarchiquement inférieur. Ces ports sont globaux pour le module et peuvent être exploités par les différentes fonctions de supervision définissant les objectifs que le superviseur est capable de réaliser.

La déclaration des fonctions de supervision se fait en leur affectant un nom (e.g. *fonction1*, fig 13) et une liste d'arguments typés (e.g. *consigne* pour fonction1, fig. 13). Ces fonctions peuvent être appelées par des superviseurs hiérarchiquement supérieurs via envoi de requêtes de configuration particulières aux superviseurs.

2.3.2.2) Implémentation des fonctions de supervision

Comme ce document l'a expliqué en section 2.2.3.2, les fonctions de supervision sont implémentées par un ensemble de règles. Ces règles sont traduites dans une structure de donnée générique qui elle-même exploite un ensemble de fonctions et de variables spécifiques. L'environnement de développement fournit un langage de description des comportements des superviseur qui permet à l'utilisateur de **déclarer ces règles** puis de les **compiler** sous la forme de ces structures de données, fonctions et variables C. La figure 14 propose un exemple de code de définition d'une fonction de supervision à partir de règles.

```
action(TypeA consigne){
local{
TypeC temp_message;
string temp_mess_type;
TypeD failure_data;
TypeA temp_consigne = consigne;
}

rules{
FIRST:
      [elapsed=1ms]
            subscribe MAA:2 =* :2;subscribe MAA:0 =* :1;
            activate schema schemaReceptionMessage()
      [MAA:2<connection_status == "disconnected">]

SECOND:
      [(MAA:2<(connection_status=="disconnected" & test("dedede",temp_consigne))>
      and started(FIRST){infinite})]
            subscribe MSC:1=>:3;
            activate schema schemaCommande1(consigne = temp_consigne)
      [MSC:1(failure_data = failure, temp_mess_type = "commutation")]

THIRD:
      [endof(SECOND)]
            notify :1(failure_message_type=temp_mess_type)
      [started(THIRD)]

FOURTH:
      [endof(SECOND)]
            compute calcul("1.24",temp_consigne.k);
            activate schema schemaCommande2(consigne = temp_consigne)
      [endof(FIRST)]

RECEPTION:
      [MAA:0(temp_message=message_received)]
            parameterize MSC(temp_message.value :1);
            activate resttime module MAD
      [started(RECEPTION)]

}
}
```

Figure 14: Exemple d'implémentation d'une fonction de supervision

Une fonction de supervision définit un ensemble de variables locales (dans le bloc **`local`**, figure 14) qui servent à stocker des données utilisées en interne par la fonction de supervision. Ces variables locales peuvent être initialisée à partir des arguments de la fonction de supervision (`temp_consigne = consigne`), ce qui sera effectif au moment de la réception de la requête de configuration représentant la demande de lancement d'un objectif. Sinon les variables locales sont initialisées par leur valeur par défaut définies à partir de leurs types respectifs.

Le comportement de la fonction de supervision à proprement parler est implémenté par un ensemble de règles (inclues par le bloc **`rules`**), qui sont chacune identifiées de manières unique par un label (e.g. label `FIRST` pour la première règle). Chaque règle est décrite avec une **précondition** (premier bloc d'une règle compris entre `[]`), une **séquence d'actions** (délimitée par les `[]` des pré et post conditions, les actions étant séparées par un `;`) et une **postcondition** (second bloc d'une règle compris entre `[]`), comme cela a été expliqué dans la section 2.2.3.2. L'ensemble des actions possibles a déjà été détaillée dans la section 2.2.3.2 et le document de prise en main de l'environnent de développement détaille l'ensemble des mot-clefs et des expressions qui permettent de définir les conditions et les actions. A titre d'exemple, un événement de module est représenté

par le nom du module suivi du caractère deux points et du numéro de port concerné (e.g. `MAA:2`, dans la postcondition de la règle 1).

Comme nous le voyons dans la figure 14, le code de l'éditeur généré à partir de la grammaire **Xtext** gère la coloration syntaxique en plus de la vérification de la conformité du code à la grammaire c'est-à-dire la vérification syntaxique « élémentaire » (i.e. vérification de la structure du code directement induite par la définition de la grammaire) ainsi qu'une vérification sémantique « basique », notamment les références (e.g. le module `MAD` référencé est inconnu dans la règle `RECEPTION`, il est souligné en rouge). A partir de la grammaire Xtext les outils associés à ce langage permettent de générer automatiquement le méta-modèle EMF correspondant à la grammaire. Ce méta-modèle est exploité par les autres langages et outils utilisés pour développer l'environnement.

Le langage **Check/Xtend** a servi a définir tout un ensemble de contraintes à respecter pour que le code écrit à travers l'éditeur, lui-même transformé en ligne en modèle EMF, soit totalement conforme aux contraintes que nous voulions rajouter au méta-modèle EMF. Il a permis d'exprimer des règles syntaxiques plus ou moins complexes (e.g. contrainte sur les noms des modules qui doivent être défini par tris caractères majuscules exactement) qui n'ont pas été définies directement dans la grammaire, ainsi que des règles sémantiques (e.g. vérification de type partout où cela est nécessaire, par exemple pour le passage de paramètre effectué dans la règle `RECEPTION`). Grâce à cet outil de génération de vérificateurs, l'environnement ContrACT fourni une coloration particulière pour les erreurs rencontrées (automatiquement sous-lignées en rouge) ainsi qu'une fenêtre faisant un rapport d'erreur.

Enfin, le langage de templates **Xpand/Xtend** nous a permis d'écrire un générateur de code à partir des informations contenues dans le méta-modèle EMF. La procédure de génération prend en entrée le modèle EMF écrit par l'utilisateur, et conforme au méta-modèle après vérification, puis interprète les fichiers templates décrits en **Xpand** pour effectuer des opérations de création de dossiers et fichiers et d'écriture dans ces fichiers.

2.3.2.3) Conception et développement de l'environnement

La figure 15 propose un résumé de la manière dont a été décrit l'environnement de développement mais également de son fonctionnement. On peut décomposer ce résumé sur trois niveau d'abstractions distincts.

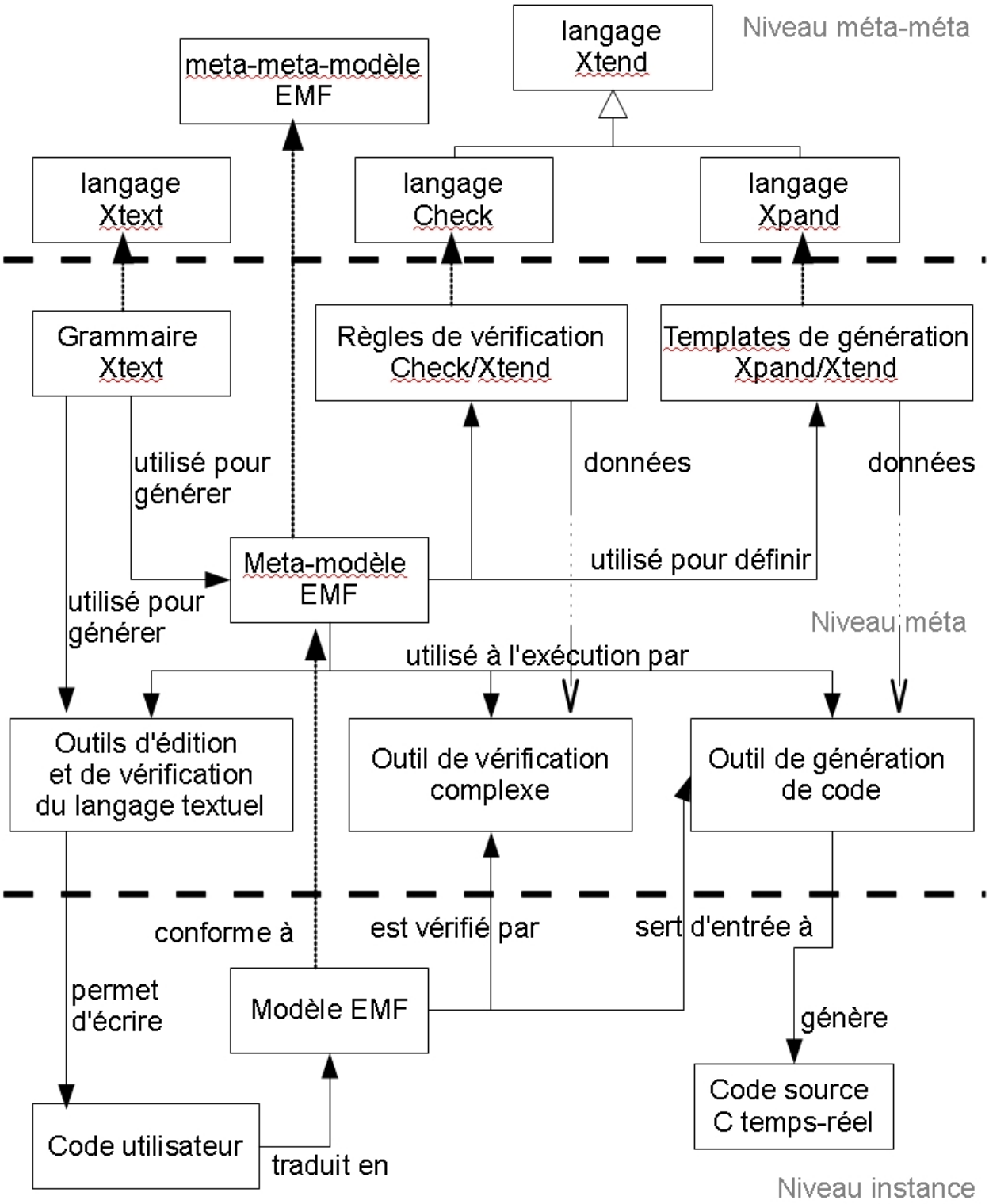

Figure 15: Conception et fonctionnement de l'outil

Le niveau **méta-méta** est celui des langages, concepts et outils **utilisés pour décrire et générer l'environnement de développement**. Ce niveau regroupe :

- le **méta-méta modèle EMF** qui permet de décrire des méta-modèle EMF
- les langages dédiés (*Xtext*, *Xtend*, *Check*, *Xpand*) à la description des différentes fonctionnalités de l'environnement (grammaire du langage dédié, règles de vérifications et règles de génération).
- Les **outils ECLIPSE** exploitant les langages et le méta-modèle pour générer le code de l'environnement.

Le niveau méta-méta est celui fourni par ECLIPSE pour générer du code suivant une approche MDE.

Le niveau **méta** est celui dans lequel nous avons travaillé pour définir le **langage dédié à la description d'architectures de contrôles** suivant la méthodologie proposée. Il s'est agit d'exploiter les langages du niveau supérieur pour définir la **grammaire** du langage dédié (*Grammaire XText*), ses **règles de vérification** (*Règles de vérification Check/Xtend*) et son **générateur de code** (*Templates de génération Xpand/Xtend*). A partir de ces artefacts logiciels nous avons utilisés les outils fourni par le niveau méta-méta pour générer:

- Le **méta-modèle EMF** à partir de la grammaire Xtext. Ce méta-modèle est le modèle de donnée qui sert de « pivot » pour tous les outils logiciels générés.
- Les **outils logiciels** d'édition (*Outils d'édition et de vérification du langage textuel*), de vérification (*Outils de vérification complexe*) et de génération (*Outil de génération de code*) générés par les outils du niveau méta-méta et exploitant le méta-modèle à l'exécution. L' *Outils de vérification complexe* exploite les **règles de vérifications** que nous avons décrit comme des données d'entrée et l' *Outil de génération de code* exploite de façon similaire les **templates de génération Xpand/Xtend**.

Le niveau **instance** est celui dans lequel on **utilise l'environnement de développement** dédié qui a été créé à partir des outils logiciels générés dans le niveau supérieur. Pour l'utilisateur il s'agit donc d'écrire du **code utilisateur dans le langage dédié** que nous avons défini, ce qui entrainera la génération en ligne d'un **modèle EMF** conforme au méta-modèle EMF que nous avons défini au niveau supérieur. Le modèle EMF est automatiquement exploité:

- par l'outil de vérification qui va le valider ou le cas échéant **reporter à l'utilisateur les erreurs** qu'il contient au regard des contraintes que nous avons exprimé au niveau supérieur.
- Par l'outil de génération qui génère le **code C temps-réel interfacé avec le middleware** conformément aux templates de génération que nous avons décrit au niveau supérieur.

L'utilisation de la méthodologie MDE et des technologies ECLIPSE correspondantes, si elle peut sembler complexe, a permis un énorme gain de temps et de qualité au développement de cet environnement. Par ailleurs, un très grand nombre d'évolutions rapides devient gérable par modification des artefacts du niveau méta et re-génération des outils (plugins ECLIPSE) correspondants, ce qui nous permet de nous adapter rapidement (dans une certaine mesure) aux éventuelles demandes des utilisateurs de l'environnement dans les projets PROSIT et ASSIST.

3) Conclusion

Ce rapport a présenté l'ensemble de la méthodologie ContrACT, développée notamment dans le cadre des projets PROSIT et ASSIST, dédiée au développement d'architectures de contrôle logicielles de robots. Ce document couvre l'ensemble de la méthodologie, depuis les concepts « haut niveau » sur la manière d'organiser une architecture de contrôle, jusqu'à la conception de l'outil logiciel support de la méthodologie, en passant par les étapes essentielles de définition d'un modèle de programmation, de son utilisation et de son implémentation dans un middleware robotique.

Les évolutions possibles de cette méthodologies auront dans le futur deux sources : les besoins exprimés par les utilisateurs et traduit (si nécessaire) aussi bien au niveau conceptuel que logiciel ; les évolutions propres issues de nos réflexions en interne sur des manquement de la méthodologie ou du middleware. Sur ce dernier point, nous envisageons pour l'instant deux types d'évolution concrètes:

- l'amélioration du **mécanisme d'ordonnancement** afin de le rendre plus configurable par l'utilisateur et plus optimisé en terme d'exploitation des capacités du processeur.
- L'enrichissement du langage dédié au niveau de la définition des **types de données**, actuellement volontairement limitée dans un souci de simplicité de développement du middleware et de l'outil. Il s'agit notamment de proposer l'utilisation de **collections** dans la définition et l'exploitation des types de données.

Ces modifications entrainerons des modifications à la fois sur le middleware et sur l'outil de développement.

4) Références

[Eljalaoui 2007] Abdellah El Jalaoui. *Gestion Contextuelle de Tâches pour le contrôle d'un véhicule sous-marin autonome.* Thèse de doctorat, Université Montpellier II , Laboratoire d'Informatique de Robotique et de Microélectronique de Montpellier (LIRMM), soutenue le 19 décembre 2007.

[Trapier 2006] Manoël Trapier. Mise en œuvre de l'architecture bas niveau d'un système de commande de robot sous marin. Mémoire Master, Université Montpellier II, Laboratoire d'Informatique de Robotique et de Microélectronique de Montpellier (LIRMM), 2006.