# Architectures logicielles pour la robotique

L. Nana

Laboratoire d'Informatique des SYstèmes Complexes (LISyC), EA3883
Université de Bretagne Occidentale
20 Avenue Le Gorgeu
C.S. 93837 – BP 809
*29238 BREST Cedex 3*

nana@univ-brest.fr

## Résumé

*Le besoin en architectures logicielles robotiques se justifie par la complexité croissante des tâches confiées aux robots. Elles permettent de structurer les différents niveaux de développements en niveaux d'abstractions et d'améliorer la réutilisabilité et la modularité des composants matériels et logiciels des systèmes robotiques. En théorie, au moins un composant d'un robot se doit d'être utilisable sur plus d'un robot. Par exemple, un contrôleur de robot marcheur pourrait également contrôler une roue motrice sur un robot à roue. Toute réutilisation de ce type nécessite, de toute évidence, un ensemble commun de spécifications. Toutes ces raisons ont conduit au développement d'architectures robotiques, mais aucune architecture dominante unique n'a émergé. L'élaboration de normes dans ce domaine est d'une importance certaine. Quelques travaux ont été effectués dans ce sens, mais le sujet reste ouvert. De même, la sûreté de fonctionnement, bien qu'ayant atteint une certaine maturité du point de vue du matériel, nécessite des solutions adaptées au niveau du logiciel, et doit être prise en compte tant au niveau des langages destinés à la programmation robotique qu'au niveau de la conception et de la mise en oeuvre des environnements de programmation robotique. La sûreté de fonctionnement logicielle est d'autant plus importante que le logiciel prend une place de plus en plus importante dans les systèmes robotiques.*

*Nous proposons dans cet article, de faire le point sur les architectures logicielles pour la robotique et d'examiner plus particulièrement la prise en compte de la sûreté de fonctionnement logicielle des applications robotiques.*

## Mots Clef

Architectures logicielles, robotique, sûreté de fonctionnement logicielle.

## 1 Introduction

Aujourd'hui, les robots sont, et effectuent des tâches, de plus en plus complexes. D'autre part, un gros projet robotique peut impliquer des centaines d'ingénieurs et/ou chercheurs et plusieurs institutions. Dans ce contexte, la disponibilité d'architectures adéquates semble indispensable afin de structurer les différents niveaux d'abstraction et d'améliorer la réutilisation et la modularité des composants matériels et logiciels des systèmes robotiques. Le logiciel prend une place de plus en plus importante dans les systèmes robotiques. L'utilisation de techniques de génie logiciel est donc nécessaire voire indispensable, dès les premières phases de réalisation d'un projet robotique, afin d'éviter les coûts de re-conception et de faciliter l'évolutivité et la maintenance des systèmes mis en oeuvre. Dans cet article, nous nous intéressons plus particulièrement à la sûreté de fonctionnement logicielle dans les architectures robotiques. En effet, les systèmes robotiques sont par essence des systèmes critiques. L'intégration et l'adaptation de mécanismes de sûreté de fonctionnement, en particulier logiciels, à ces systèmes, est donc d'un intérêt indéniable.

Dans la deuxième section de cet article, un bref aperçu des architectures robotiques est présenté. La troisième section est quant à elle consacrée à l'utilisation de mécanismes de sûreté de fonctionnement du logiciel dans le cadre d'applications robotiques. La quatrième et la cinquième sections relatent deux expériences en matière de conception et de réalisation d'architectures logicielles pour des applications robotiques sûres, à savoir l'architecture associée au langage PILOT (Programming and Interpreted Language Of actions for Telerobotics) et une architecture de réalisation de mission de l'IFREMER. La première a été conçue et mise en oeuvre au sein du LISyC et s'applique principalement à la robotique mobile. La deuxième est quant à elle relative aux applications robotiques sous-marines. Cet article se termine par une conclusion en sixième section.

## 2 Bref aperçu des architectures robotiques

La communauté robotique reconnaît qu'aucune architecture n'est parfaite pour répondre à toutes les

tâches, et que différentes tâches ont différents critères de succès qui conduisent à différentes architectures. Les architectures de programmation robotique peuvent être regroupées en quatre grandes catégories:

- les architectures centralisées classiques
- les architectures hiérarchiques,
- les architectures comportementales et
- les architectures hybrides.

Les premiers travaux concernant les architectures de contrôle robotique étaient inspirés de l'intelligence artificielle [27], c'est-à-dire organisés autour de processus décisionnels et d'un état symbolique du monde et du robot. Les architectures conçues suivant cette philosophie font partie de la catégorie des *architectures centralisées classiques*. Elles placent la planification au centre du système et partagent l'axiome suivant lequel le problème central en robotique est la cognition, c'est-à-dire, la manipulation de symboles pour maintenir et agir sur un modèle du monde, le monde étant l'environnement avec lequel le robot interagit. Parmi les architectures centralisées, nous pouvons citer: le système de planification STRIPS du robot Shakey [28] dans lequel le plan est statique et le monde supposé inchangé au cours de l'exécution du plan, les architectures Blackboard [18] qui accumulent des données sur le monde et prennent des décisions immédiates basées à la fois sur les objectifs à priori variables et un monde changeant. Pour une tâche donnée, si le système peut modéliser le monde suffisamment bien, et si le monde obéit à son (ses) modèle(s), et si le système peut récupérer l'information pour l'intégrer dans le cœur de la planification centrale, alors une architecture centralisée classique constitue un bon choix pour la réalisation de la tâche. Les architectures centralisées conviennent bien aux tâches pour lesquelles la réactivité et le réflexe ne sont pas des critères essentiels.

Les *architectures hiérarchiques* décomposent la programmation des applications en niveaux de plus en plus abstraits. Chaque niveau a pour rôle de décomposer une tâche que lui a recommandée le niveau supérieur, en tâches plus simples qui seront ordonnées au niveau inférieur. Le niveau le plus haut gère les objectifs globaux de l'application, alors que le niveau le plus bas commande les actionneurs du robot. L'instance la plus connue de ce type d'architecture est NASREM (Nasa/nbs Standard REference Model) [24]. Dans la même famille, nous pouvons citer l'architecture du LIFIA [17] et l'architecture SMACH de l'I3S (Informatique, Signaux et Systèmes de Sophia-Antipolis) [35]. Les architectures hiérarchiques telles que NASREM font encore l'hypothèse que le meilleur moyen d'interagir avec le monde est à travers la manipulation et le raisonnement au sujet de modèles du monde, bien qu'elles reconnaissent qu'il doit y avoir différents modèles du monde pour raisonner au sujet des différents aspects du monde. Ce que peut réaliser un tel système, en utilisant des modèles prédictifs du monde, c'est une très haute précision. Chaque couche a un modèle de ce qui va se produire dans le monde, étant donné un ensemble d'entrées et de sorties, et il revient aux couches inférieures de s'assurer que ce qui était attendu se réalise précisément. Les architectures hiérarchiques sont appropriées pour des tâches qui s'exécutent dans un environnement prévisible et qui requièrent une haute précision. Leur principal défaut est la taxonomie des modules du système, a priori imposée artificiellement, qui sert à les restreindre plutôt qu'à les supporter. En effet, la façon dont chaque module dans le système est structuré n'est pas définie par les besoins de la tâche, mais par l'endroit où il s'insère dans l'architecture. Les architectures hiérarchiques ont généralement une réactivité assez faible: compte tenu de la décomposition systématique de la programmation, la chaîne allant des capteurs aux actionneurs en passant par les processus décisionnels capables de répondre à des changements de l'environnement est complexe, entraînant des temps de réponse longs.

Les architectures comportementales sont nées au milieu des années 80 avec l'architecture "subsomption" proposée par Brooks [5]. Elles sont issues de l'observation de comportements animaux simples, et sont basées sur l'idée qu'un comportement complexe et évolué d'un robot peut émerger de la composition simultanée de plusieurs comportements simples. Brooks définit un comportement élémentaire comme "un traitement prenant des entrées capteurs et agissant sur les actionneurs". L'architecture DAMN (Distributed Architecture for Mobile Navigation) proposée par Rosenblat [31] à l'Université de Carnegie Mellon est une autre variante des travaux de Brooks. Les travaux de Brooks ont mis en évidence l'atout de ce type d'architecture: la rapidité de la réaction du système face aux événements extérieurs ou à des situations spécifiques. Les architectures comportementales ont fait leurs preuves dans de nombreuses et parfois spectaculaires expérimentations concernant la robotique mobile, car la réactivité qui les caractérise permet d'aborder la navigation dans un environnement dynamique. Toutefois, la complexité des applications reposant sur cette approche va rarement au delà de la navigation. En effet, plusieurs comportements sont souvent en concurrence pour le contrôle des actionneurs et on ne peut pas, à priori, assurer la stabilité d'exécution de la loi de commandes complexes telles que celles requises pour le contrôle de bras manipulateurs. Ces approches présentent une autre limitation: les comportements étant préétablis, le système s'accommode difficilement d'un changement de mission impromptu.

Face aux lacunes des deux précédentes catégories d'architectures, certains chercheurs ont proposé des architectures hybrides qui allient les capacités réactives des architectures comportementales et les capacités de raisonnement propres aux architectures hiérarchiques. Ces architectures peuvent être suffisamment souples et puissantes pour que leur domaine d'utilisation en terme de variété de robots contrôlés et de type d'application justifie

leur commercialisation. Parmi elles, nous pouvons citer: le CONTROLSHELL [33] vendu par la société californienne RTI (Real-Time Innovations), l'architecture du LAAS [ALA 98] qui a fait ses preuves dans les domaines de la robotique mobile, que ce soit sur les plates-formes HILARE ou dans l'expérience MARTHA, ou encore l'architecture ORCCAD (Open Robot Controller Computer Aided Design system) de l'INRIA [6][19]. L'architecture ORCCAD a la particularité d'être indépendante du système à piloter. Elle autorise également la spécification et la validation de missions en robotique.

# 3 Sûreté de fonctionnement dans les architectures robotiques

## 3.1 Bref aperçu des mécanismes de sûreté de fonctionnement

La sûreté de fonctionnement est une propriété importante aux différents niveaux du processus de contrôle et de commande tant en ce qui concerne la télérobotique, qu'au regard des systèmes automatisés de production. Elle touche aux différents aspects de la réalisation d'une mission, partant de la conception du plan à son exécution sur le système commandé, en passant par le processus d'interprétation du plan ou de génération de l'exécutable.

Deux approches principales sont souvent utilisées pour la mise en oeuvre de la sûreté de fonctionnement: la prévention des erreurs et la tolérance aux fautes [22][20][21]. La prévention des erreurs vise à écarter à priori les fautes et les erreurs qui mettent en cause la fiabilité du système, et ceci avant toute utilisation régulière de ce dernier. Pour atteindre cet objectif, les principaux moyens sont l'utilisation de méthodes et de langages de spécifications formels et l'utilisation de tests. La tolérance aux fautes est quand à elle basée sur le principe suivant lequel la prévention des erreurs, bien que bénéfique, ne permet pas de garantir une élimination totale des erreurs dans le système. Elle a pour but de permettre au système de se comporter de façon satisfaisante même en présence de fautes.

## 3.2 Solutions pour la sûreté de systèmes

La prise en compte de la sûreté de fonctionnement logicielle dans la conception d'architectures robotiques reste encore marginale de nos jours. Un certain nombre de travaux ont toutefois été effectués dans ce domaine. Zalewski et *al*. [38] ont souligné la complexité des solutions actuellement fournies pour la vérification des systèmes de contrôle informatiques qui restreint leur applicabilité à des systèmes simples, alors que la complexité des applications critiques est habituellement élevé et continue à s'accroître drastiquement avec les progrès des technologies informatiques. Ils ont étudié deux approches principales.

La première approche est basée sur l'« Analyse d'Arbres de Fautes » (AAF) et l' « Analyse de Mode de Défaillance et d'Effet » (AMDE). Il s'agit de techniques d'analyse de sûreté utilisées avec succès dans des systèmes conventionnels (non basés sur l'informatique). Elles sont utilisées lors de la conception du système et se focalisent sur les conséquences de défaillances des composants. Des adaptations ont été proposées pour l'analyse des systèmes de logiciels sûrs [23]. Zalewski et *al* ont proposé une méthode d'analyse informelle de sûreté de systèmes basés sur le logiciel utilisant l'AAF [38]. Une application à l'industrie nucléaire a été effectuée [25]. L'avantage de cette solution est que les techniques sous-jacentes sont déjà bien utilisées pour de nombreuses applications industrielles, ce qui permet aux ingénieurs de sûreté de s'adapter facilement à leurs nouvelles versions. L'inconvénient est que ces techniques sont pour la plupart plutôt informelles. Une adaptation de ces solutions aux logiciels orientés objet a également été proposée par Zalewski et al. Elle fait l'hypothèse que les modèles orientés objets des composants logiciels sont fournis avec leurs spécifications formelles. Cette approche a été appliquée à une étude de cas de contrôle de feux de circulation ferroviaire.

La deuxième approche est basée sur l'utilisation de méthodes formelles et semi formelles et de modèles initialement développés pour le domaine logiciel: logique temporelle, réseaux de Petri, LOTOS, modèles action-événements, etc. Zalewski et *al* ont combiné dans un seul système intégré, via une interface commune, des outils d'ingénierie traditionnels tels que ceux reposant sur UML, avec des outils de méthodes formelles tels que des outils de « model checking » ( Statecharts, etc.) [2].

Garbajosa et *al* ont proposé et mis en oeuvre un outil pouvant servir comme partie frontale pour le test des systèmes et acceptant des descriptions de tests en langage naturel, afin d'affranchir les ingénieurs du test de la nécessité d'avoir une parfaite maîtrise des systèmes physiques pour lesquels les tests sont définis et des techniques de programmation, qui leurs sont peu familiers [10].

Rutten a proposé une trousse à outils pour la programmation sûre d'applications robotiques [32]. Cette dernière est basée sur la synthèse de contrôleurs [30].

Différents autres travaux basés sur l'utilisation des techniques d'intelligence artificielle ont été effectués pour le diagnostic de faute [39] et la supervision [13].

## 3.3 Mise en œuvre dans les architectures robotiques

Seabra Lopes et *al* proposent dans [34], une architecture pour l'assemblage de tâches qui fournit à différents niveaux d'abstraction, des fonctions pour

l'ordonnancement des actions, le contrôle de leur exécution, le diagnostic et le recouvrement d'erreur. La modélisation des défaillances d'exécution faite à travers des taxonomies et des réseaux de causalité joue un rôle central dans le diagnostic et le recouvrement.

Dans l'architecture de subsomption les comportements sont modélisés chacun par une (ou plusieurs) machine(s) à états finis augmentée(s). Cette modélisation permet l'application de méthodes de vérification, mais aussi la mise en œuvre de mécanismes de tolérance aux fautes par l'exploitation des suppresseurs et des inhibiteurs.

Dans l'architecture du LAAS le niveau décisionnel est réactif aux comptes-rendus d'exécution des niveaux inférieurs. Ces comptes-rendus peuvent être exploités pour la mise en œuvre d'actions de recouvrement.

L'architecture ORCCAD de l'INRIA est une des architectures qui mettent un accent sur la sûreté des applications [36]:

- Exécution temps réel rigoureuse des lois de commande.
- Utilisation du langage synchrone ESTEREL pour la spécification de la partie contrôle.
- Utilisation des outils de vérification formelle pour la partie contrôle des applications.

L'aspect sécuritaire dans la réalisation de missions robotiques a également été l'une des motivations principale du projet « Architecture Logicielle pour la robotique mobile et téléopérée » qui a conduit à la création du langage PILOT et de son architecture logicielle. Dans la section suivante, nous présentons les travaux réalisés dans ce contexte pour la sûreté de fonctionnement des applications robotiques. Une brève description de l'architecture de contrôle du langage PILOT est d'abord effectuée. L'approche de sûreté et les solutions mises en œuvre dans le cadre de cette architecture sont ensuite abordées.

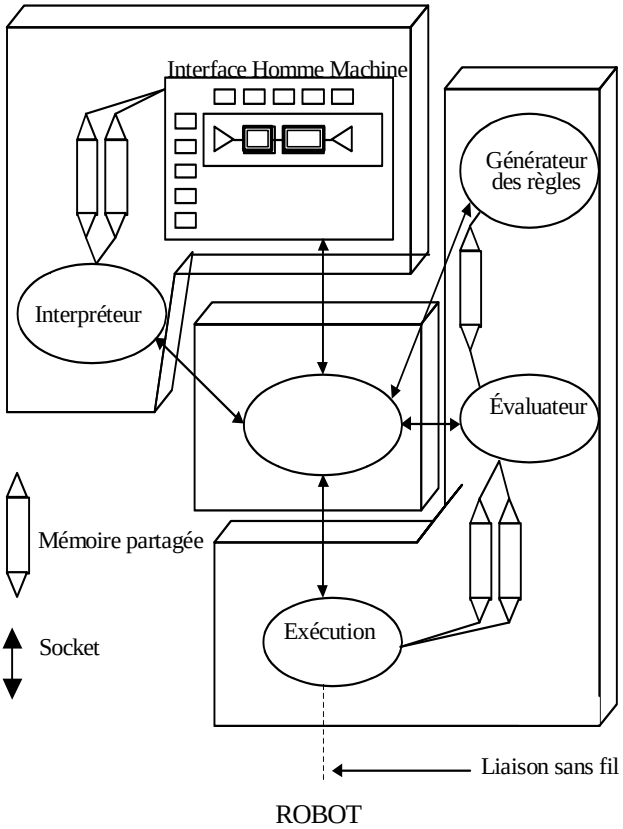


FIG. 2 – Système de contrôle PILOT

# 4 Mécanismes de sûreté de l'architecture PILOT

## 4.1 Architecture logicielle PILOT

Le système de contrôle de PILOT (FIG. 1) est l'interface entre l'utilisateur et la machine pilotée (Robot Cible) [9] [26]. Il comporte six modules: une *Interface Homme Machine* (IHM), un *Serveur de Communication*, un *Générateur de Règles*, un *Evaluateur*, un *Module d'Exécution* ou *Driver* et un *Interpréteur*. Ces modules sont exécutés en parallèle et communiquent par socket et par mémoire partagée. Le système de contrôle peut s'exécuter soit en mode centralisé, soit en mode distribué. Le choix du mode d'exécution est effectué de façon statique (avant la compilation). L'*IHM* fournit des moyens pour la construction de plans, la création dynamique d'actions (sans recompilation du code), et la modification du plan avant et au cours de l'exécution de ce dernier. Elle intègre également des moyens pour la supervision de l'exécution du plan. L'*IHM* stocke le plan dans une zone de mémoire partagée avec l'*interpréteur*. Ce dernier lit le plan en mémoire partagée et envoie des ordres (demande d'évaluation de précondition d'une action, ordre de démarrage de l'action, etc.) aux autres modules afin de réaliser l'exécution du plan. Le *serveur de communication* gère les communications inter modules. Le rôle du *générateur de règles* est de transformer les chaînes de caractères des règles de préconditions et de surveillance en arbres binaires. Il stocke le résultat dans une zone de mémoire partagée avec l'*évaluateur*. Ce dernier évalue les règles de précondition et de surveillance à partir des arbres binaires correspondants. Le *module d'exécution* réalise l'interface entre le robot et le système de contrôle. Il traduit les ordres de haut niveau du plan en ordres de bas niveau compréhensibles par la machine téléopérée. Le module d'exécution supporte différents protocoles de communication (connexion série, Ethernet, FDDI).

## 4.2 Sûreté de fonctionnement avec PILOT

### *4.2.1 Mécanismes internes*

Les actions PILOT comportent des **règles de précondition et de surveillance.** Ces règles constituent des moyens de sûreté pour l'application PILOT. En effet,

une action ne peut être exécutée que si sa précondition est vraie. De même, lorsqu'une règle de surveillance est satisfaite, le traitement associé est effectué (le traitement par défaut est l'arrêt de l'action). Ce mécanisme est équivalent au mécanisme des exceptions et constitue une solution pour la mise en oeuvre de la tolérance aux fautes. Si nous considérons par exemple l'action *avancer* pour un robot mobile équipé de détecteurs d'obstacles, une règle de précondition pourrait être le test d'absence d'obstacle. L'une des règles de surveillance serait par exemple le test de présence d'obstacle avec comme traitement associé l'arrêt de l'exécution de l'action.

Les plans PILOT sont modifiables au cours de leur exécution, ce qui est un atout majeur pour la tolérance aux fautes. En effet, l'opérateur peut, en cas de dysfonctionnement dans l'exécution d'un plan, apporter des modifications permettant au système de revenir dans un état de fonctionnement satisfaisant (poursuite de la mission ou arrêt dans un état sûr).

La nature interprétée du langage et la possibilité de modifier des plans en cours d'exécution rendent possible l'exécution de plans incomplets. On peut ainsi lancer l'exécution d'un plan sans fin de séquence principale ou contenant une structure parallèle dont l'exécution ne peut se terminer en l'état parce qu'elle est incomplète. Afin de pallier cet inconvénient, l'environnement de contrôle de PILOT a été doté d'un mécanisme d'**édition dirigée par la syntaxe** permettant de garantir la validité syntaxique du plan à chaque phase de sa construction (insertion, modification, suppression de primitives). Cette approche permet de préserver les avantages de la possibilité de modifier dynamiquement des plans: terminaison de plans bien assurée, etc.

L'édition dirigée par la syntaxe ne prend pas en compte la validité sémantique du plan lors de sa modification au cours de l'exécution. Une **approche basée sur le formalisme de « synthèse des contrôleurs »** a été incorporée dans l'architecture de contrôle afin de sécuriser les modifications en cours d'exécution, notamment par la gestion d'aspects tels que la suppression, l'insertion ou la suppression de primitives. Grâce à ce travail, il est désormais possible d'effectuer des actions de compensation ou de recouvrement d'erreurs « sécurisées » en cours de mission.

### *4.2.2 Mécanismes liés au processus de développement*

Les différents modules du système de contrôle du langage PILOT ont été modélisés à l'aide d'**automates d'états finis** et des **algorithmes d'interprétation** ont été définis pour les différentes primitives du langage. Ces éléments fournissent une bonne base pour la prévention d'erreurs (application de méthodes de vérification formelle). Ils permettent en outre d'éviter des erreurs dues à la distorsion de l'information tout au long du processus de développement logiciel.

Afin d'augmenter la robustesse du système PILOT des approches de **tests statique et dynamique** ont été appliquées à son interpréteur qui est l'un de ses modules les plus critiques. La nature réactive des applications robotiques augmente la complexité des opérations de test, car l'on doit, en plus des facteurs usuels, prendre en compte les événements difficilement maîtrisables générés par le robot. Un autre point important est la prise en compte des dommages éventuels que peuvent engendrer des tests effectués directement sur le robot. Un simulateur de robot simple a donc été construit pour les opérations de test.
Le *test statique* a consisté, d'une part, en la lecture du code source dans le but de détecter les erreurs de programmation et, d'autre part, en l'analyse du code source par rapport aux algorithmes d'interprétation et la sémantique de PILOT.
Le *test dynamique* a, quant à lui, consisté en la définition de jeux de tests et en leur application au code binaire de l'interpréteur. Les données de test ont été définies en combinant une approche fonctionnelle au retour d'expérience des tests déjà effectués. Pour la définition de l'échantillon représentatif des données de test, nous avons adopté une approche incrémentale. La séquence vide a d'abord été testée, puis les autres primitives du langage ont été testées individuellement. Trois combinaisons des primitives du langage ont ensuite été considérées:

- *Combinaison en longueur* par l'accroissement du nombre d'éléments dans les séquences du plan.
- *Combinaison en largeur* par l'accroissement du nombre de branches dans le parallélisme, la préemption ou la conditionnelle.
- *Combinaison en profondeur* par l'accroissement du niveau d'imbrication.

Afin d'avoir un ensemble borné de jeux de tests, nous avons émis un ensemble d'hypothèses: les actions du même type sont interchangeables, l'ensemble des séquences résultant de la combinaison de paires quelconques de primitives est représentatif de l'ensemble des séquences comportant deux ou plus de primitives excepté pour les problèmes de mémoire, etc.
Chacune des approches de test appliquées à l'interpréteur a permis de détecter des erreurs de différentes natures (erreur de conception dans la gestion des interruptions logicielles et dans la gestion de la terminaison des actions continues, erreurs de programmation, etc.). Ces erreurs ont été corrigées et très peu de dysfonctionnements ont été observés dans l'utilisation de l'interpréteur depuis ce travail.

Bien que les techniques de test statiques et dynamiques décrites ci-dessus se soient révélées très utiles dans la détection et la correction d'erreurs dans les programmes d'interprétation, leur utilisation ne permet pas de garantir la conformité de l'interprétation des plans à la sémantique

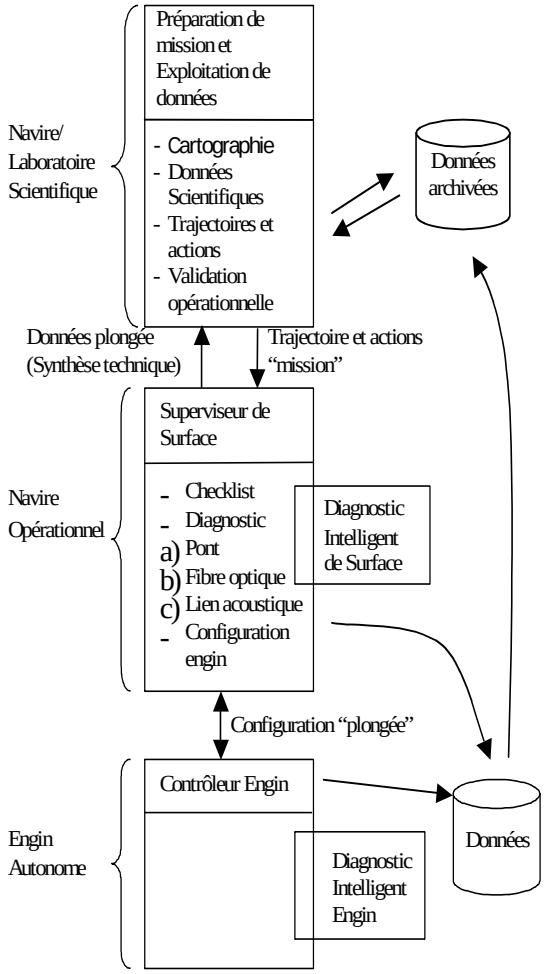


FIG. 2 – Architecture globale

opérationnelle du langage PILOT. Un travail complémentaire a été fait pour pallier cet inconvénient. Il a consisté à **modéliser les algorithmes d'interprétation** et à **vérifier leur conformité par rapport à la sémantique opérationnelle du langage** afin de corriger les éventuels dysfonctionnements et de régénérer le code de l'interpréteur à partir du modèle validé. Les réseaux de Petri colorés (RdPC) ont été utilisés pour la modélisation et la vérification. Le support logiciel utilisé a été Design/CPN (http://www.daimi.aau.dk/DesignCPN).

Les RdP et plus particulièrement les RdP colorés ont été choisis pour différentes raisons. Leur nature graphique offre la convivialité souhaitée dans le but d'utiliser ultérieurement le modèle comme médium de communication entre les différentes personnes impliquées dans le développement du logiciel de contrôle, afin d'éviter des erreurs dues à la distorsion de l'information. Ils permettent de représenter relativement simplement les différents concepts de l'algorithmique et de la programmation. La disponibilité d'outils, pour la simulation et la vérification des modèles, a également été un critère important.
La modélisation a permis de constater que des simplifications sont envisageables tant au niveau de la représentation interne d'un plan, qu'au niveau des algorithmes d'interprétation, et d'appliquer ces dernières au système de contrôle. Des plans de tests ont été générés en s'appuyant sur l'approche adoptée au cours du test dynamique des algorithmes d'interprétation. La simulation de l'exécution de ces plans a permis de détecter des problèmes de terminaison. Cette dernière ne permettant d'explorer, dans la pratique, qu'une partie des chemins d'exécution, un travail complémentaire a été effectué. A partir des RdPC modélisant les algorithmes d'interprétation et d'un plan de test, le graphe des marquages accessibles correspondant aux chemins d'exécutions possibles est généré à l'aide de l'outil Design/CPN. Le graphe des marquages et le plan de test sont ensuite transmis au programme de vérification qui examine, pour chacun des chemins d'exécution, la satisfaction de la sémantique opérationnelle du langage. Une extension de l'environnement Design/CPN a été effectuée pour intégrer notre programme de vérification.

Après cette présentation des mécanismes de sûreté offerts par l'environnement PILOT et des approches de tests et de vérification adoptées pour renforcer sa robustesse, nous présentons, dans la section suivante, l'étude et l'intégration de mécanismes de sûreté de fonctionnement dans une architecture globale de préparation, de supervision et d'exécution de missions d'engins sous-marins autonomes (Fig. 2.). Ce travail s'est effectué en collaboration avec la Division Robotique de l'IFREMER Toulon. Nous présenterons les solutions proposées pour la sûreté ainsi que les propositions faites pour leur mise en œuvre aux différents niveaux de l'architecture.

## 5. Mécanismes de sûreté pour des missions d'engins sous-marins autonomes

### 5.1 Niveau préparation de mission

Deux approches sont abordées pour la sûreté: la vérification de propriétés et la tolérance aux fautes. Nous les abordons dans les sous-sections qui suivent.

#### *5.1.1 Vérification de propriétés*

Il s'agit de vérifier l'adéquation entre les contraintes issues de la spécification de la mission et les caractéristiques de l'engin et de l'environnement (accessibilité de la zone à explorer, précision des capteurs et de la trajectoire, fréquence d'acquisition des mesures, durée de la mission, énergie, limites de vitesse et d'altitude, capteurs et charges utiles adaptées à la mission, temps de calibration, maintien - si nécessaire - de l'engin dans la zone à explorer), de vérifier la cohérence, en terme d'enchaînement d'actions et de logique d'exécution de la mission spécifiée par le scientifique (par exemple, certaines exécutions ne peuvent être faites en parallèle, les post-conditions d'une action et les préconditions de l'action suivante peuvent être incompatibles), et d'effectuer des diagnostics sur l'engin, le système de

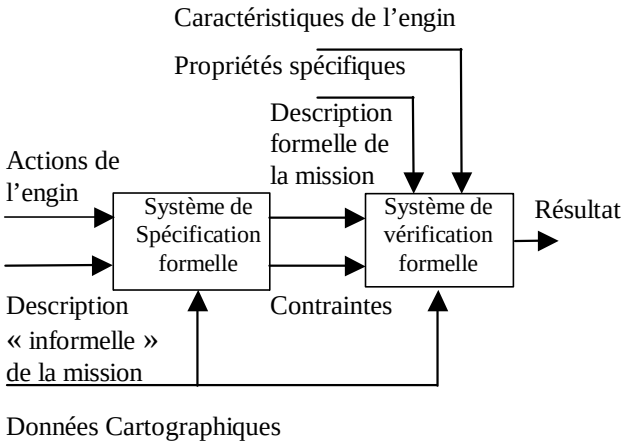


FIG. 3 – Système de spécification et de vérification

contrôle, et leurs modèles éventuels (les données des plongées précédentes pourront être utilisées pour ajuster certains paramètres du diagnostic). La phase de diagnostic pourra permettre, par exemple, de vérifier la terminaison de la mission.
La figure 3 montre la structure du sous-système de spécification et de vérification proposé à ce niveau.

Différents outils sont envisageables pour la conception et la spécification. Nous pouvons citer l'environnement STOOD de TNI-Europe qui permet par ailleurs de générer des programmes pour différents systèmes de vérification. Au niveau de la vérification, les approches de « model-checking » et de démonstration peuvent être explorées. Différents outils sont disponibles. Certains outils s'appuient sur l'approche synchrone très utilisée pour la conception de systèmes réactifs dont font partie les systèmes robotiques: outils commerciaux tels que SCADE et ESTEREL [8], SILDEX [11], environnement CELL CONTROL spécialisé pour les automatismes industriels de ATHYS [12]. D'autres formalismes graphiques de spécification et de vérification, synchrones et dédiés au contrôle commande tels que STATECHARTS [16], SYNCCHARTS [3] et GRAFCET [1], ou asynchrones tels que les Réseaux de Petri, offrent également des possibilités intéressantes.
L'approche de démonstration a également donné lieu à différents outils de preuves (prouveurs) [14]: Isabelle [29], HOL [15], Coq [7], PVS [37], Boyer-Moore [4]. La plupart de ces systèmes utilisent des logiques d'ordre supérieur qui sont extrêmement souples et expressives.

### *5.1.2 Tolérance aux fautes*

Il s'agit ici de prendre en compte les défaillances envisageables afin de permettre, au niveau de la définition de la mission, de prévoir des actions de recouvrement.

L'environnement de conception de plans de missions devra fournir des moyens permettant d'intégrer les réactions aux défaillances (mécanismes de recouvrement).

Dans l'élaboration du plan de mission, on pourra prévoir 2 cas de figures:

- Introduction de la redondance pour pallier certaines défaillances, par exemple modification de la trajectoire initiale suite à la détection d'un obstacle. Il s'agit dans ce cas de créer un plan de « repli » pour la partie jugée critique. La gestion d'un tel aspect dépend également de la richesse du système de contrôle qui peut déjà être équipé d'un mécanisme automatique de contournement d'obstacle.
- Classement des actions par niveau de « criticité » de façon à prendre les actions de compensation appropriées en cas de dysfonctionnement (abandon et passage à l'action suivante, abandon de la mission, saut à une action spécifique ou à un point particulier du plan de mission, …).

L'approche proposée pour l'intégration de mécanismes de traitement d'erreur consiste, après construction du *plan de mission standard* (c'est-à-dire ne prenant pas en compte la gestion de fautes autres que celles spécifiées lors de la création des actions), à spécifier pour chaque action ou primitive le traitement de fautes correspondant. La primitive est sélectionnée et les conditions de fautes, ainsi que les réactions associées sont définies. Le système de spécification se sert alors du plan et des données saisies pour générer le fichier de plan de mission incorporant le traitement de fautes.
Au niveau des actions, les conditions et traitements de fautes initiaux de l'action sont étendus en cas de besoin pour prendre en compte de nouvelles conditions de fautes et leur associer les traitements souhaités. Les conditions de fautes sont des expressions logiques basées sur des valeurs de capteurs, les états d'exécution d'actions ou de primitives et les états de fautes reçus du système de contrôle « bas niveau ».
Les réactions associées aux conditions de fautes sont: l'arrêt de l'action ou de la primitive (il faudra tenir compte du caractère interruptible ou non de l'action) et/ou le saut vers un point du plan et/ou l'exécution d'une séquence à définir.

## 5.2 Niveau superviseur de surface

Deux aspects sont considérés à ce niveau, la **supervision de la mission** et la **vérification de propriétés** relatives au déroulement de la mission.

Il s'agit, pour la supervision de mission, de récupérer des informations relatives au déroulement de la mission et de les rendre disponibles, par affichage à l'écran et éventuellement stockage dans un fichier accessible par l'utilisateur, afin de permettre à l'opérateur de prendre des décisions notamment en cas de dysfonctionnement. Les données récupérées sont les valeurs de capteurs, les états d'exécution des actions, les états de défaillances éventuelles (alarmes, etc.).

En ce qui concerne la vérification de propriétés, un *Système de Diagnostic Intelligent* (SDI) permet d'effectuer des vérifications plus élaborées sur le déroulement de la mission. Il est formé de modules de diagnostic intelligent ayant chacun sa spécificité (par exemple une technique particulière d'intelligence artificielle ou la gestion d'un type de fautes particulier), et d'un module de décision. Le module de décision est chargé, d'une part, de collecter les informations utiles au diagnostic et de les transmettre aux modules de diagnostic intelligent appropriés, et, d'autre part, d'effectuer la synthèse des informations de diagnostic reçues des différents modules de diagnostic intelligent afin de transmettre, aux modules le requérant (par exemple, le système contrôlé ou un autre module réalisant par exemple une trace des défaillances), les informations sur les anomalies détectées ou les ordres de correction.

Les différents diagnostics et recouvrements envisagés au niveau des SDI sont les suivants:

- Diagnostic de défaillance des effecteurs,
- Diagnostic de défaillance des liens de communication,
- Contrôle des batteries (autonomie),
- Contrôle de la précision de la trajectoire,
- Contrôle de la précision/qualité des données mesurées,
- Contrôle du logiciel de contrôle embarqué (cas de défaillance partielle).

Pour la défaillance totale du logiciel de contrôle embarqué, des procédures de recouvrement sont prédéfinies. De même, le contrôle de l'ordinateur de surface et de son logiciel est assuré par l'opérateur.

Le diagnostic de défaillance des effecteurs des AUVs peut être considéré comme un problème de classification ordinaire. La disponibilité d'expertises humaines et d'exemples oriente cependant vers le choix d'une solution basée sur le mixage des approches symboliques et neuronales. En ce qui concerne le recouvrement de ces défaillances, il n'existe pas pour l'instant de base d'exemples. Il s'agit d'un problème de contrôle, pour lequel la théorie automatique classique ne peut être appliquée, dont le recouvrement est plus lié aux stratégies de contrôle qu'aux approches de contrôle. Etant donné que la sélection des stratégies de contrôle peut dépendre de plusieurs contraintes quelquefois difficiles à mesurer ou exprimer, la théorie de contrôle de la logique floue semble être une solution intéressante.

Le diagnostic des liens de communication et le contrôle de la batterie de l'AUV sont des problèmes similaires. Ils correspondent davantage à des problèmes de gestion de risques qu'à des méthodes de diagnostic pur. Par conséquent, l'objectif n'est pas uniquement d'effectuer un diagnostic par l'étude d'une situation statique donnée, mais au contraire d'observer l'évolution de cette situation dans le temps, et puis d'évaluer le risque d'avoir une défaillance ou une situation de conflit. Les outils basés sur des méthodes probabilistes tels que les réseaux Bayésiens sont particulièrement bien adaptés à ce type de problème.

## 5.3 Niveau contrôleur d'engin

Le sous-système de sûreté au niveau contrôleur engin comporte un **Système de Diagnostic Intelligent** dont le principe est le même que celui du niveau Supervision de Surface, un **gestionnaire de modes**, un **gestionnaire de fautes**, un **gestionnaire d'énergie**, un **système d'archivage** et un **module de conversion**. Un protocole robuste est utilisé pour le transfert de la mission entre la surface et l'engin, afin de se prémunir contre toute corruption de données et contre l'exécution de missions incomplètes. La mise à jour du plan doit être possible à tout moment à travers une liaison acoustique (mode immergé), radio ou télémétrique (modes surface et « sous surface »). Contrairement au SDI de Surface, dont le rôle est celui d'un agent chargé d'analyser / diagnostiquer les fautes, de fournir à l'opérateur une synthèse des résultats des diagnostics et, éventuellement, de lui proposer des actions à entreprendre pour pallier les défaillances (l'opérateur est seul chargé d'envoyer les ordres d'actions correctives à l'engin), le *Système de Diagnostic Intelligent de l'Engin* a un fonctionnement autonome. Sur la base des diagnostics qu'il aura effectués, il proposera directement les actions correctives au Système de Contrôle de l'Engin. Le *gestionnaire de mode* contrôle l'ensemble des transitions d'états du véhicule. Le *gestionnaire de fautes* détecte les fautes du véhicule et prend les actions sur fautes prédéfinies. Les réponses sur fautes possibles sont: « stop et surface », « change l'étape de la mission », « ignore la faute et continue ». Le *gestionnaire d'énergie* gère l'énergie du passé et prévoit l'énergie du futur. Lorsque l'énergie atteint certains niveaux de « commutation », un événement d'avertissement et un événement d'alarme sont déclenchés. Le *sous-système d'archivage* mémorise les données relatives à la mission. Il transmet au Système de Diagnostic Intelligent Engin (SDIE) des informations lui permettant de vérifier la cohérence des données et de détecter d'éventuelles anomalies. Le *module de conversion* se charge de convertir les actions de recouvrement demandées par le SDIE en une séquence adaptée aux modules destinataires, et de convertir les informations émanant des autres modules dans un format adapté au SDIE.

## 6. Conclusion

Malgré l'existence d'architectures telle que NASREM initialement conçue comme standard pour la robotique mobile, l'étude des architectures logicielles existantes relève la nécessité de poursuivre les efforts de normalisation dans ce domaine. L'on peut noter l'utilisation d'un certain nombre de techniques du génie logiciel dans la conception et la réalisation d'architectures logicielles robotiques: structuration en couche, notion de composants logiciels et de modularité. Un certain nombre d'architectures orientées objets ont également été conçues.

La notion d'agents est également exploitée, en particulier dans le cadre d'application multi-robots.

En ce qui concerne la sûreté de fonctionnement logicielle, la majorité des travaux porte sur le diagnostic et le traitement de fautes à l'aide de techniques d'intelligence artificielle. Seules quelques architectures logicielles intègrent ou ont donné lieu à l'utilisation de méthodes formelles. De même, les mécanismes de tolérance aux fautes logicielles sont peu exploités. L'utilisation croissante de la programmation distribuée dans ces architectures, nécessite pourtant d'incorporer des mécanismes tels que la réplication très souvent prise en compte au niveau matériel. L'on peut également noter le peu de report d'expériences en matière de tests rigoureux dans la réalisation de ces architectures qui, bien que dû peut-être à la perception même des activités de tests, traduit un manque d'intérêt en la matière. Ces activités de test sont pourtant très importantes dans le processus de développement de logiciels sûrs.

De façon globale, les éléments susmentionnés renforcent la nécessité d'un effort dans l'application de techniques de génie logiciel, et plus particulièrement de celles relatives à la sûreté de fonctionnement, à la mise en oeuvre d'architectures robotiques.

Les travaux réalisés dans le cadre de la conception de l'architecture logicielle PILOT ont apporté des solutions génériques pour la sûreté de fonctionnement d'applications robotiques: mécanisme d'édition dirigée par la syntaxe, recouvrement d'erreur à travers la possibilité de modification de missions en cours d'exécution, mécanisme de sécurisation de modifications en cours d'exécution. Les approches de tests statique et dynamique et les méthodes formelles (modélisation et vérification à l'aide de RdP colorés) appliquées à l'interpréteur de plans peuvent également s'appliquer à d'autres environnements de programmation de mission. Dans le cadre de l'étude de mécanismes de sûreté pour l'architecture de programmation de missions d'AUV une approche pour la gestion des fautes été proposée. L'idée novatrice dans cette approche est la hiérarchisation de la gestion des fautes, et la spécification par extension qui permet, d'une part, d'étendre la gestion initiale de fautes intégrée à l'action, évitant ainsi la redondance des traitements, et, d'autre part, d'avoir une gestion de fautes associée à chaque primitive. Les solutions proposées aux différents niveaux de l'architecture globale de préparation de mission sont applicables à d'autres environnements similaires voire à des applications robotiques dans des domaines non maritimes (robotique mobile terrestre ou manufacturière).

## Références


[1] ADEPA, *Le Grafcet*, Cépaduès Editions, Paris, France, 1992.

[2] Al-Daraiseh, A., Zalewski, J. and Toetenel, H. Software engineering in ground transportation systems. In *Proceedings of the SCI'01, 5th world multiconference on systemics, cybernetics and informatics.* Orlando, FL., July, 2001.

[3] André C., *Representation and analysis of reactive behaviors: A synchronous approach*, In CESA'96, IEEE-SMC, Lille, France, 1996.

[4] D.J.B. Bosscher, I. Polak and F.W. Vaandrager. *Verification of an audio control protocol*. In H. Langmaak, W. P. de Roever and J. Vytopil, editors. Proceedings of the third School and Symposium on Formal Techniques in Real-Time and Fault-Tolerant Systems, volume 863 of Lecture Notes in Computer Science, pages 170-192, Springer-Verlag.

[5] Brooks R. A., *A robust layered control system for a mobile robot*, IEEE Journal of Robotics and Automation, pages 14-23, Mars 1986.

[6] Castillo E., D. Simon, B. Espiau and K. Kapellos, *Computer-aided design of a generic robot controller handling reactivity and real-time control issues*, Rapport de recherche 1801, INRIA, November 1992.

[7] Devillers M.C.A., W.O.D. Griffioen, J.M.T. Romijn and F.W. Vaandrager. *Verification of a leader election protocol: formal methods applied to IEEE 1394*. Report CSI-R9728, Computing Science Institute, Nijmegen, 1997.

[8] Dima C., A. Girault, C. Lavarenne, and Y. Sorel. *Off-line real-time fault-tolerant scheduling*. In 9th Euromicro Workshop on Parallel and Distributed Processing, PDP'01, pages 410-417, Mantova, Italy, février 01.

[9] Fleureau J.L., **L. Nana** Tchamnda, L. Marcé and L. Abalain, *Remote-controlled vehicle using PILOT Language*, In ANS'99, Pittsburgh, Pennsylvania, American Nuclear Society, 1999.

[10] Garbajosa J., O. Tejedor and M. Wolff. Natural language front end to test systems. Annual review in Automatic Programming, vol. 19, pp. 261-267, 1994.

[11] Girault A. *Sur la répartition de programmes synchrones*. Thèse de Doctorat, INPG, Grenoble, France, Janvier 1994.

[12] Girault A. *Elimination of redundant messages with a two pass static analysis algorithm*. Parallel computing, 28(3):433-453, mars 2002.

[13] Gomez P., S. Romero, P. Serrahima and I. Alarcon, A real time expert system for continuous assistance in process control: a successful approach, Annual Review in Automatic Programming, vol. 19, pp. 371-375, 1994.

[14] Groote J. F., F. Monin and J.C. Van de Pol. *Checking verification of protocols and distributed systems by computer*. In D. Sangiorgi and R. de Simone, Proceedings of Concur'98, Sophia Antipolis, LNCS 1466, pages 629-655, Springer-Verlag, 1998.

[15] Goldschlag D. M. *Verifying safety and liveness properties of a daily insensitive fifo circuit on the Boyer-Moore prover*. International Workshop on Formal Methods in VSLI Design, 1991.

[16] Harel D., *STATECHARTS: A visual approach to complex systems*, Science of Computer Programming, 8(3), 231-274, 1987.

[17] Hassoun M. and C. Laugier, *Reactive motion planning for an intelligent vehicle*, In Intelligent Vehicles'92 Symposium, pages 259-264, Detroit, july 1992.

[18] Hayes-Roth B., *A blackboard architecture for control*, Artificial Intelligence, 26:pp. 251-321, 1985.

[19] Kapellos K., D. Simon and B. Espiau, *Control laws, tasks, procedures with ORCCAD; application to the control of an underwater arm*, In 6th IARP (International Advanced Robotic Program), La Seyne sur Mer, France, 1996.

[20] Kermarrec Y., L. Nana and L. Pautet, *Implementing recovery blocks in GNAT: a powerful fault tolerance mechanism and a transaction support*, In ACM, editor, Proceedings of the TRI-Ada'95 Conference, Anaheim, California, Novembre 1995.

[21] Kermarrec Y., L. Nana, L. Pautet, « Providing fault-tolerant services to distributed Ada 95 applications », In ACM, editor, *Proceedings of the Tri Ada'96 conference*, Philadelphia, USA, Décembre 1996.

[22] Laprie J. C., « Sûreté de fonctionnement des systèmes informatiques et tolérance aux fautes: concepts de base », *TSI*, 4(5):419-429, Septembre 1985.

[23] Leveson, N., Cha, S.S., Shimeall, T. J., 1991. Safety and verification of Ada programs using software fault trees. IEEE Software 8(7), 48-59, 1991.

[24] Lumia R., J. Fiala and A. Wavering, *The NASREM robot control system and testbed*, IEEE Journal of Robotics and Automation, 5(1), pp. 20-26, 1990.

[25] Maier T., FMEA and FTA to support safety design of embedded software in safety-critical systems. In *Proceedings of the ENCRESS conference on safety and reliability of software based systems.* Belgium, 1995.

[26] Nana Tchamnda L., J.L. Fleureau and L. Marcé, *A control system for PILOT: software architecture and implementation issues*, ANS'01, ANS 9th International Topical Meeting on Robotics and Remote Systems, Seattle, Washington, March, 2001.

[27] Nilsson N., *A mobile automation: an application of artificial intelligence techniques*, In Proc. Int. Joint Conf. on Artificial Intelligence, pp. 509-520, 1969.

[28] Nilsson N., *Shakey the robot*, Technical Report 323, SRI, Menlo Park, CA.

[29] Paulson L. C. *On two formal analyses of the Yahalom protocol*. Technical report 432, Computer Laboratory, University of Cambridge, 1997.

[30] Ramadge P. J., Wonham W. M., « The control of discrete event systems », *Proceedings of the IEEE*, Special issue on dynamics of discrete event systems, vol. 77, no. 1, pages 81-98,1989.

[31] Rosenblatt J., *DAMN: A distributed architecture for mobile navigation*, Journal of Experimental and Theoretical Artificial Intelligence, 9(2/3), pp. 339-360, 1997.

[32] Rutten E., A framework for using discrete control synthesis in safe robotic programming », Rapport de recherche, INRIA, 2000.

[33] Schneider S., V. Chen, G. Pardo-Castellote and H. Wang, *ControlShell: A software architecture for complex electro-mechanical systems*, International Journal of Robotics Research, Special issue on Integrated Architectures for Robot Control and Programming, 1998.

[34] Seabra Lopes L. and L.M. Camarinha-Matos, Learning to diagnose failures of assembly tasks, Annual Review in Automatic Programming, vol 19, pp. 97-103, 1994.

[35] Tigli J.Y., *Vers une architecture de contrôle pour robot mobile orientée comportement, SMACH*, Thèse de Doctorat, Université de Nice - Sophia Antipolis, Janvier 1996.

[36] Turro N., *MaestRo: Une approche formelle pour la programmation d'applications robotiques*, Thèse de doctorat, Université de Nice - Sophia Antipolis, Septembre 1999.

[37] Vitt J. and J. Hooman. *Assertional specification and verification using PVS of the Steam Boiler Control System*. In J.-R. Abrial, et al., editors, Formal Methods for Industrial Applications: Specifying and Programming the Steam Boiler Control. Volume 1165 of Lecture Notes in Computer Science, 1996.

[38] Zalewski, J., W. Ehrenberger, F. Saglietti, J. Gorski & A. Kornecki, Safety of computer control systems: challenges and results in software development, Annual Review in Control, vol. 27, pp. 23-37, 2003.

[39] Zhang J., A. J. Morris and G. A. Montague, Fault diagnosis of a cstr using fuzzy neural networks, Annual Review in Automatic Programming, vol 19., pp. 153-158, 1994.