Gestion Contextuelle de Tâches pour le contrôle d'un véhicule sous-marin autonome

Abdellah El Jalaoui

Université Montpellier II

SCIENCE ET TECHNIQUES DU LANGUEDOC

Thèse

En vue de l'obtention du grade de

Docteur De L'Université De Montpellier II

Discipline : **Génie Informatique, Automatique et Traitement du Signal**
Formation Doctorale : **Systèmes Automatiques et Microélectroniques**
Ecole Doctorale : **Information, Structures, Systèmes**

Présentée et soutenue publiquement par

Abdellah EL JALAOUI

Le 19 Décembre 2007

Gestion Contextuelle de Tâches pour le contrôle d'un véhicule sous-marin autonome

Jury

Rapporteurs	Massimo	CACCIA	*Senior Researcher, CNR - ISSIA*
	Simon	LACROIX	*Directeur de Recherche, CNRS - LAAS*
Examinateurs	Michel	PERRIER	*Ingénieur de Recherche, IFREMER*
	Philippe	FRAISSE	*Professeur, Université Montpellier II*
	Lionel	LAPIERRE	*Chercheur contractuel, CNRS - LIRMM*
Co-Directeur de thèse	David	ANDREU	*Maître de Conférences, Université Montpellier II*
Directeur de thèse	Bruno	JOUVENCEL	*Professeur, Université Montpellier II*

Thèse préparée au LIRMM Laboratoire d'Informatique, de Robotique et de Microélectronique de Montpellier (CNRS), 161, rue Ada 34392 Montpellier Cedex 5

... et le savoir augmente lorsqu'il est partagé.

Ali Ibn Abi-Taleb.

à ma Mère

Avant-Propos

Les travaux présentés dans ce manuscrit ont été effectués au Laboratoire d'Informatique, de Robotique et de Microélectronique de Montpellier au sein du groupe Robotique Sous-Marine. Je tiens à remercier mon Directeur de thèse Mr. Bruno Jouvencel pour sa confiance et la liberté d'action qu'il m'a accordées.

Je tiens à remercier tout particulièrement mon Co-Directeur de thèse Mr. David Andreu pour sa confiance, son soutien, la qualité de son encadrement et surtout pour ses qualités humaines. J'exprime également ma gratitude à Messieurs Massimo Caccia et Simon Lacroix pour avoir accepté d'être rapporteur sur les travaux présentés. Je suis reconnaissant envers Messieurs Michel Perrier, Philippe Fraisse et Lionel Lapierre pour avoir accepté de prendre part au jury.

Ce travail est le fruit d'une collaboration scientifique et humaine riche et c'est dans ce sens que j'adresse de sincères remerciements à Olivier Parodi, Vincent Creuze, Lionel Lapierre, Manoël Trapier et Stéphane George.

Je souhaite remercier Mr. Etienne Dombre pour avoir soutenu ma candidature pour cette thèse, Mr. Philippe Fraisse et Mr. David Guiraud pour leurs conseils au moment de la rédaction.

Ces travaux au LIRMM, m'ont donné l'occasion de pouvoir apprécier l'amitié de certaines personnes que je ne saurais oublier : Ashvin Sobhee, Meziane Bennour, Philippe Amat, Jean-Mathias Spiewak, Yousra Benzaïda, Nicolas Riehl, Baptiste Magnier, Ibrahim Ben Lazreg, Robin Passama, Tomás Salgado Jimenez, Nabil Badereddine, Cyril Montagna, Pierre Desinde.

Je tiens à remercier tous ceux que j'ai pu connaître et apprécier pendant ces années de thèse : Aurélien Thouvenot, Iskander Tagiev, Lounis Douadi, Arrahmane, Mohammed Ben Abdellah, Abdessalam Yassine, Roger Pissard-Gibollet et Soraya Arias de l'INRIA-Rhônes-Alpes, Ahmed Bakari, Rachid Aboutayeb, Alex Ngouanga, Waël Gouja, Nouredine Razine, Delphine Colson, Abdellah Makhchan, Nizar Ben Regba, Mohammade Najim, Iqbol Jon.

Je ne saurais terminer sans rendre hommage aux membres de ma famille qui ont patienté tout ce temps et qui m'ont soutenu et supporté : mes parents, Fadma, Mina et Ali El Jalaoui, Fatima Aalouane, Abdellah Loumaid et Mohammed Oubihi.

Table des matières

Table des figures

Introduction

Employé pour la première fois par l'écrivain tchèque Karel Čapek [1] dans la pièce de théâtre R.U.R (Rossum's Universal Robots) en 1920, le mot "robota" dont dérive le mot "robot" signifie dans les langues slaves : "travail d'esclave". Les premiers robots se sont plutôt bien prêtés à cette définition en remplaçant l'homme dans toutes les tâches pénibles et répétitives. Déroulant des tâches simples dans des conditions hostiles comme la peinture des carrosseries automobiles sous atmosphère toxique, les robots se sont très vite imposés dans le secteur industriel. Sans aucun pouvoir de décision, ces machines exécutent aveuglément, dans un environnement de topologie constante ou prédictible, des séquences de mouvements préprogrammées par l'opérateur humain.

Les progrès technologiques, notamment en informatique et en électronique, permettent aux robots de s'équiper de capteurs pour sonder leur environnement et d'unités de calcul pour analyser les situations et prendre des décisions. Le besoin de substitution de l'homme par le robot dans les missions d'intervention en milieu hostile ou inaccessible (démantèlement de centrale nucléaire, inspection d'épave de bateau, exploration spatiale) a amené la robotique à évoluer vers plus d'autonomie. Contrairement à son étymologie d'esclave docile, le robot acquiert plus de pouvoir de décision et interagit avec un environnement souvent inconnu ou en constante évolution. Il est défini par Bernard Espiau [2] comme étant : "*...une machine agissant physiquement sur son environnement en vue d'atteindre un objectif qui lui a été assigné. Cette machine est polyvalente et capable de s'adapter à certaines variations de ses conditions de fonctionnement. Un robot est doté de fonctions de perception, de décision et d'action. Il possède des capacités de mouvement propres et peut entrer en interaction avec les objets de son environnement. Il a en outre la capacité de coopérer à divers degrés avec l'homme.*"

Plus généralement, on entend par robot tout système conçu par l'homme pour le remplacer dans ses tâches, avec parfois une morphologie ou un comportement rappelant celui de l'homme (ou de l'animal).

Les objectifs de conception des robots ont eux aussi évolué en fonction des possibilités technologiques et surtout des besoins sociétaux : de la précision et de la rapidité pour les premiers dispositifs robotiques destinés à opérer sur une chaîne de production (soudage, manutention, assemblage...), de plus en plus d'autonomie décisionnelle pour la robotique d'intervention et enfin, avec l'émergence d'une robotique dite "domestique", le robot se dote d'une intelligence artificielle et va jusqu'à exprimer des sentiments [3].

En robotique sous-marine, domaine central de notre étude, l'autonomie décisionnelle

prend une place importante dans la conception des véhicules d'exploration. On peut distinguer deux grandes classes de robots sous-marins : les engins reliés physiquement à la surface (par exemple les ROV-Remotly Operated Vehicle) et les engins autonomes (par exemple les AUV-Autonomous Underwater Vehicles). La première catégorie d'engins représente une technologie éprouvée et largement utilisée tandis que la seconde est encore peu développée. Cependant, un certain nombre de calculs comparatifs réalisés par des professionnels du domaine (compagnies pétrolières et développeurs d'engins) font toujours ressortir un net avantage à l'usage des AUV [4]. L'absence de liaison physique avec la surface, permettant l'apport d'énergie et l'échange fiable de données entre l'opérateur et le véhicule, est la principale difficulté rencontrée dans la conception et le contrôle des AUV. En effet, le milieu sous-marin est un milieu en constante évolution dont les propriétés physiques ne permettent pas au robot de garder un contact de qualité avec l'opérateur en surface (faibles débits des communications acoustiques). On ne peut donc compter sur le fait de transmettre en permanence (en ligne) au robot l'itinéraire à suivre, ni lui communiquer, au cours de la mission, les tâches à effectuer. A partir d'une mission décrivant un ensemble d'opérations à mener à bien à des endroits géographiques donnés, le véhicule devra alors, de lui-même, analyser le contexte pour trouver l'itinéraire à emprunter et exécuter les actions demandées selon la situation. Dans ces conditions, le contrôleur du robot, c'est à dire l'organe qui se charge de commander la partie opérative (instrumentation), joue un rôle déterminant dans la réussite de la mission et dans la sauvegarde de l'intégrité du système. Centre de décision du robot, il doit analyser l'état du véhicule (données capteurs, temps écoulé) ainsi que les objectifs à atteindre pour construire l'ensemble des commandes à appliquer aux actionneurs et remplir ainsi la mission.

Cette application logicielle qu'est le contrôleur du robot est d'autant plus complexe que le robot est doté d'un grand nombre d'instruments de mesure et que les tâches à effectuer demandent une adaptation à la situation. Cette complexité se retrouve dans la conception de logiciels pour la robotique sous-marine. En effet, les missions effectuées par les robots sous-marins, et notamment les AUV, nécessitent l'utilisation d'une instrumentation embarquée conséquente. Cette instrumentation se subdivise en deux catégories : l'une comprenant l'ensemble des instruments de bord servant à la navigation même de l'appareil (centrale inertielle, GPS, gouvernes, propulseur ...), l'autre regroupant le matériel (mesure de salinité, relevé topologique ...) qui va servir à la (aux) tâche(s) scientifique(s) de la mission (recherche de source d'eau douce, inspection de pipeline ...).

La forte absorption des ondes électromagnétiques par l'eau est l'une des propriétés intrinsèques du milieu sous-marin. Elle engendre des difficultés au niveau de la communication avec les engins ainsi qu'au niveau de leur capacité à scruter l'environnement. L'alternative consiste à utiliser des capteurs à base d'ondes acoustiques. Outre la baisse de performance, due notamment à la lenteur relative de la propagation de l'onde sonore, des problèmes d'interférence entre capteurs travaillant à des fréquences proches ou de valeurs multiples l'une de l'autre, apparaissent et peuvent rendre les informations enregistrées très bruitées, voire inexploitables. Recrutant les capteurs au fil de la mission, selon les tâches à effectuer ou par nécessité résultante des contraintes de terrain (détection et évitement d'écueil), le contrôleur du véhicule devra alors de lui-même gérer l'utilisation des différents capteurs pour éviter les conflits dus aux interférences et organiser en conséquence son plan d'actions.

Pour assurer une autonomie décisionnelle convenable au véhicule, le logiciel de bord

devra déterminer à chaque moment les tâches (au sens robotique du terme), à traiter et veiller à la bonne utilisation de l'instrumentation. Si l'on ajoute à ces contraintes de fonctionnement les contraintes de développement comme l'hétérogénéité des domaines de compétence de ses maîtres d'oeuvre (automatique, traitement du signal, traitement d'image, informatique industrielle, informatique, électronique, mécanique,...), le contrôleur logiciel apparaît alors comme un organe complexe.

La conception de telles applications requiert une méthodologie, une décomposition, une structuration, etc... mettant en exergue les différentes briques logicielles avec une spécification claire de la "place" et du rôle de chacune. On parle alors d'architecture logicielle. Cette architecture doit spécifier sans ambiguïté chaque entité informatique du logiciel, son fonctionnement et ses différentes interactions tant au sein de l'application de contrôle qu'avec le matériel. Elle doit répondre à un ensemble de critères (ou propriétés) tels que la modularité, la réactivité, l'évolutivité, ... desquels dépendent son adéquation aux contraintes de commande d'un véhicule robotisé, ainsi que sa capacité à évoluer tant au regard du progrès technologique (nouveaux instruments) qu'au regard des avancées scientifiques (nouveaux modèles de commande par exemple).

La solution architecturale proposée, en plus des critères évoqués, doit être simple et accessible à tous les membres de nos projets scientifiques (selon leur rôle et intérêt scientifique). C'est une dimension importante dans un thème, une équipe, pluridisciplinaires. Pouvoir tester simplement une nouvelle loi de commande, intégrer/modifier rapidement l'instrumentation pour répondre aux différentes missions, tester des algorithmes d'ordonnancement, décrire simplement des missions pour des "non-roboticiens", font partie des "objectifs".

Cette proposition d'architecture doit être accompagnée des environnements et outils logiciels permettant son exploitation, et bien sûr son utilisation sur notre flotte de robots sous-marins. D'ailleurs, elle doit soutenir la mise en oeuvre de flottille de véhicules marins hétérogènes.

Plusieurs architectures ont été proposées dans la littérature pour la robotique en général, mais aucune ne s'est réellement imposée. Un grand nombre de systèmes robotiques autonomes ont été développés ces vingt dernières années notamment aux Etat Unis, mais la plupart d'entre eux sont dédiés à des tâches spécifiques et ne sont pas interopérables. Cela a poussé différents organismes à ériger des standards (Joint Architecture for Unmanned Systems - JAUS [5], Object Management Group - OMG, Open and Interoperable Autonomous Systems - OISAU) qui vont de la spécification des communications jusqu'aux méthodes de construction des architectures [6].

Notre travail ne se situera pas sur le plan méthodologique, mais sur un plan architectural au sens où la proposition issue de cette étude sera une architecture vérifiant la contrôlabilité jusqu'au plus bas niveau.

La première partie de ce mémoire sera consacrée à une étude bibliographique. Dans cette partie, le premier chapitre présente le milieu sous-marin, ses propriétés qui vont expliquer les différents choix technologiques pour l'exploitation de ce milieu, les enjeux qui vont justifier la nécessité et quelques uns des objectifs de la recherche dans ce domaine et enfin, les différents moyens mis en oeuvre en robotique pour atteindre ces objectifs. Au chapitre 2, nous présenterons les différentes architectures logicielles en robotique, ainsi que les grands principes sur lesquels ces dernières se sont basées. Au chapitre 3, on abordera certaines notions d'ordonnancement de tâches informatiques. Les logiciels

embarqués ayant des traitements dépendant fortement du temps, l'étude des principes de gestion de l'affectation de l'unité de calcul aux processus nous permettra d'apporter des solutions aux conflits entre instruments de mesure (interférences).

Une proposition d'architecture logicielle pour un robot sous-marin sera détaillée dans la deuxième partie de ce manuscrit. Le premier chapitre exposera une proposition d'architecture fondée sur des briques de bases appelées "modules". Au deuxième chapitre, nous expliquerons comment ces modules devront être composés et hiérarchisés, notamment pour établir les différents flux de contrôle et de données, pour construire une architecture. Enfin, le troisième chapitre traitera des contraintes bas niveau de l'architecture, notamment le respect des dates imposées par les lois de commande ainsi que des problèmes liés à la gestion de l'instrumentation (capteurs interférents) ; une solution basée sur un ordonnanceur sera proposée.

La troisième et dernière partie, sera consacrée aux expérimentations menées sur les engins sous-marins du LIRMM. Le premier chapitre sera dédié à une présentation détaillée des deux véhicules sous-marin Taipan II et Taipan 300 ainsi que de leurs architectures matérielles (ensemble des capteurs et actionneurs) et informatiques (carte processeur et périphériques). Dans le deuxième chapitre, nous détaillerons la préparation de l'environnement informatique de travail nécessaire à la construction de l'architecture. Nous détaillerons notamment l'implémentation de notre middleware, similaire à une infrastructure de création et d'exécution des modules de l'architecture. Dans le troisième chapitre nous présenterons les modules que nous avons construits pour l'architecture qui a été intégrée dans les véhicules du laboratoire. Enfin, un quatrième chapitre exposera les résultats des essais, menés en environnement lacustre, avec le véhicule Taipan 300.

Première partie

Etat de l'art

Chapitre 1

La robotique sous-marine

1.1 Introduction

"*La terre est bleue comme une orange*" (Paul ELUARD, L'Amour la poésie (1929)), Ce vers tiré du recueil L'Amour la poésie est une figure de style à la manière de Paul Eluard, poète du XXiéme siècle qui nous évoque la rondeur de notre planète et sa couleur dominante, le bleu. La terre doit son bleu à l'eau qui recouvre les 71% de sa surface. Cette immensité aqueuse a toujours suscité la curiosité, et a été au fil des siècles le berceau de maintes légendes et exploits humains et technologiques.

Ainsi, des hommes comme Y.P.G. le Prieur, J.Y Cousteau, J. Piccard et D. Walsh, ont écrit l'histoire contemporaine de l'exploration sous-marine en relevant des défis jusqu'alors considérés comme impossibles. Suivant les traces de G. de Lorena qui, en 1535, plongea sur deux épaves de galères romaines, après avoir réalisé la première cloche de plongée, d'après un design de L. Da Vinci. En 1776, D. Brushnell réalisa la "Tortue", le premier sous-marin d'attaque inaugurant ainsi ce qui deviendra la longue histoire de l'exploitation militaire des profondeurs océaniques. En 1960, J. Piccard et D. Walsh posa le bathyscaphe Trieste au fond des fosses des Mariannes, le point le plus profond connu du plancher océanique (10916m.), confirmant ainsi que la vie est partout. Et plus récemment, J. Cousteau a consacré sa vie à l'étude de la diversité océanique, laissant à l'humanité un héritage qui a suscité de nombreuses vocations pour reprendre le flambeau de l'exploration sous-marine.

Ces hommes ont démontré la faisabilité des interventions sous-marines, et ont circonvenu l'ensemble des verrous scientifiques et technologiques que revêtent de telles missions. Le principal objectif de ces missions était bien sûr d'assurer la survie de l'opérateur embarqué à bord des engins. Retirer l'humain du système immergé lève de nombreuses contraintes critiques, mais pose le problème de l'autonomie des engins et de leurs capacités réelles. Les Marines Nationales ont été les premières à démontrer leur intérêt pour le développement de systèmes marins autonomes. Ainsi, en 1866, R. Whitehead, pour le compte de la marine Autrichienne, a développé une nouvelle arme pour les batailles navales. Il démontra l'efficacité d'un engin flottant autopropulsé transportant une charge offensive. La classe des torpilles était née. Les missions de cet engin ne requéraient pas de contrôle ou de système de navigation sophistiqué. Il était autonome comme une balle de

revolver peut l'être.

En fait, les premiers véhicules sous-marins autonomes (Autonomous Underwater Vehicle - AUV) ont été développés durant les années soixante et soixante-dix avec le SPURV (Self-Propelled Underwater Research Vehicle, USA) et l'Epaulard (Ifremer - France). SPURV déplaçait 480 Kg et pouvait opérer à 2.2 m/s sur une durée de 5.5 heures à une profondeur de 3000m. Ce véhicule, acoustiquement relié à la surface, permettait d'effectuer des mesures physico-chimiques (salinité et température) le long de sa trajectoire, dans le cadre de la modélisation des phénomènes océaniques. Epaulard pesait 3 tonnes et pouvait naviguer à 6000m de profondeur durant 7 heures à environ 1 noeud (0.51 m/s). Ces systèmes ont été les précurseurs des quelques 2400 engins sous-marins autonomes qui sont actuellement opérés quotidiennement de par le monde [7].

1.2 Le besoin et les techniques d'exploration du milieu sous-marin

1.2.1 Exploitations industrielles, militaires et scientifiques de la mer

Offshore pétrolier

L'offshore pétrolier qui représente environ le tiers des exploitations, est principalement situé dans une partie peu profonde (100 à 200m) des zones côtières, appelée plateau continental. Pour trouver de nouveaux gisements les compagnies pétrolières doivent s'éloigner davantage des côtes au-delà du plateau continental. Dans ces zones éloignées, l'exploitation des fonds marins devient très difficile.

En effet comme le montre la figure 1.1, le plateau continental se poursuit après une brusque rupture de pente sur une zone très abrupte appelée talus continental. A la base du talus on atteint des profondeurs comprises entre 2000 et 6000 mètres formant le bassin océanique.

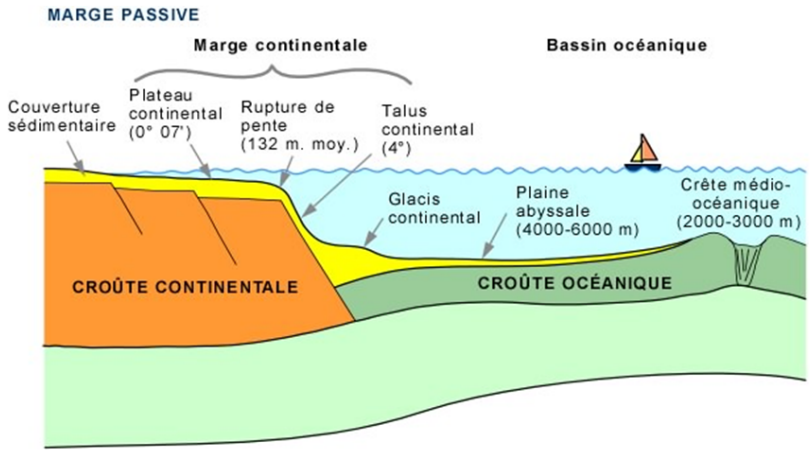


Figure 1.1 – Coupe de la zone côtière

On estime que 44 % des ressources de pétrole et de gaz exploitables par les installations pétrolières offshore se trouvent à des profondeurs supérieures à 400 mètres [4]. Cependant, la mise en place de ces installations dans de telles profondeurs nécessite au préalable des études d'exploitabilité. Ces études basées sur des analyses topographiques et des reconnaissances sismiques nécessitent de pouvoir faire descendre les appareils de mesures à proximité du fond pour obtenir des images d'assez bonne résolution.

Minerai

L'extraction de minerai dans les fonds océaniques est un défi majeur pour la prochaine décade. Par exemple à proximité des îles Hawaï, sur le plancher de l'océan Pacifique, on estime qu'il y a une étendue de 2000 milliards de tonnes de nodules [1] de manganèse. Le géologue Jean-Claude Michel qualifie les ressources minières sous-marine de "gisement inépuisable" [8]. En effet au sein des dorsales et des fossés océaniques où le fonctionnement des systèmes hydrothermaux entraîne la précipitation de métaux qui, en s'accumulant, forment de véritables gisements polymétalliques en constante génération.

Il classe ces ressources en trois principales catégories :

- les nodules polymétalliques (Co, Fe, Mn, Ni et Cu) présents sur la plupart des fond marins, reposant sur le plancher sédimentaire,
- les encroûtements à cobalt et oxydes de manganèse et de fer qui se rencontrent dans tous les océans, entre 400 et 4 000 mètres de profondeur, sur les flancs et les sommets des volcans sous-marins ainsi que sur les dorsales et les plateaux.
- les sulfures massifs sur lesquels se focalise actuellement l'industrie minière se présentent comme économiquement prometteurs.

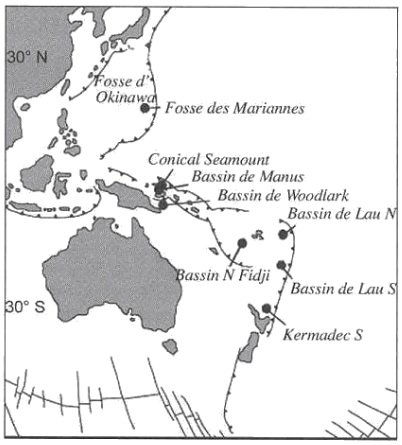


Figure 1.2 – Localisation des zones fertiles connues [8]

Les besoins dans ce domaine sont similaires à ceux de l'industrie pétrolière, excepté que la dispersion du minerai sur de vastes surfaces nécessite aux dispositifs d'exploitation de pouvoir être en mouvement [9].

1. petites concrétions constituées de métaux qu'on trouve au fond des océans.

Télécommunications

Les télécommunications sont un autre secteur de l'industrie offshore qui évolue rapidement. Ses 16% de parts d'activité [9] sont en progression. C'est en 1851 que le tout premier câble sous-marin a été posé entre Douvres et le cap Gris-Nez. Chaque année, près de 200.000 kilomètres de câbles sous-marins sont posés à travers toutes les mers du globe. Ils ont pour tâche d'acheminer le trafic téléphonique, la vidéo et les données. On assiste à une demande grandissante relative à la sécurité, la robustesse et la haute capacité des réseaux de télécommunication internationaux. Pour répondre à cette demande, les industries de télécommunication impliquées dans le déploiement offshore doivent mettre en place, inspecter et réparer les liens physiques de communication sous-marins.

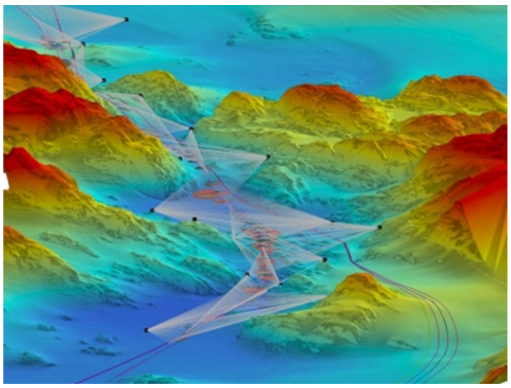


Figure 1.3 – Etude bathymétrique avant la pose d'un câble

La dépose des câbles sous-marins nécessite au préalable l'étude topographique (voir figure 1.3) de la zone concernée pour déterminer le trajet idéal, le plus court, sans risque pour le câble, en évitant les reliefs et recherchant les terrains relativement meubles puisque lorsque cela est possible, les câbles sont enterrés à deux mètres de profondeur.

Ces opérations nécessitent à plusieurs niveaux le déploiement d'engins d'interventions sous-marines :

- reconnaissance bathymétrique du trajet de pose du câble,
- ensouillage [2] des câbles (figure 1.4),
- inspection et réparations éventuelles.

Militaire

Depuis la première guerre mondiale, les opérations militaires sous-marines sont devenues d'une importance capitale dans la gestion de nombreux conflits à travers le monde. Les applications militaires les plus étudiées concernent la détection et la destruction de mines sous-marines. Certains de ces engins explosifs, vestiges de la seconde guerre mondiale, peuplent les routes de navigation maritime ainsi que les côtes atlantiques.

Ces opérations sont très dangereuses car elles nécessitent un dispositif opératoire pour approcher les zones incriminées pour détecter et détruire les objets suspects.

2. Enfouissement d'une canalisation sous-marine (oléoduc, câble) dans le sol marin, après creusage d'une souille

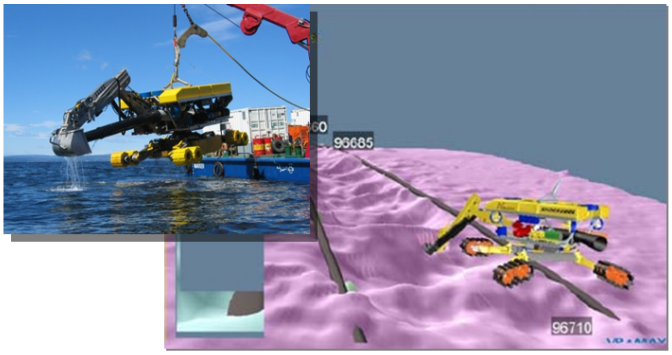


Figure 1.4 – Ensouillage d'un câble par le robot Spider développé par Nexans

Une autre activité en pleine émergence est la surveillance des ports. Pour éviter l'intrusion de plongeurs ou encore de véhicules sous-marins ennemis, il est nécessaire de quadriller régulièrement les zones sensibles comme les entrées des ports.

Environnement

L'application des lois relatives aux restrictions de pêche ainsi que la protection de l'environnement maritime requiert la détection, l'identification, la poursuite et l'interdiction des bateaux suspects. Ces procédures requièrent des informations précises et fiables pour pouvoir être prises en compte par les autorités judiciaires.

Recherches scientifiques

L'état des régions polaires est un élément indicateur de l'évolution du climat de la planète. Les études menées dans ces régions nécessitent l'inspection de zones sous-marines situées sous la banquise. L'hostilité de ces zones froides et peu accessibles encourage les recherches à s'appuyer sur des dispositifs ne faisant pas intervenir l'homme.

1.2.2 Contraintes et limitations des techniques actuelles

Les opérations en mer impliquant l'homme sont très difficiles et coûteuses à réaliser du fait du niveau de sécurité à maintenir dans pareilles conditions. L'être humain n'est pas naturellement doté pour évoluer dans les milieux aqueux. L'absence d'oxygène d'une part et les fortes pressions d'autre part réduisent les temps d'intervention des véhicules sous-marins habités.

1.2.3 Apport de la robotique dans ce domaine

La robotique sous marine va remplacer la main, puis l'intelligence humaine pour pouvoir effectuer les travaux à moindre coût et aller plus loin dans les capacités d'exploitation. L'utilisation de robots sous-marins pour intervenir dans les zones hostiles à l'homme réduit les niveaux de sécurité à la sauvegarde de l'intégrité de certains appareils de mesures onéreux.

Plusieurs types de robots sous-marins ont été développés à cet effet.

1.3 Les robots sous-marins

On classifie généralement les robots sous marins en fonction de la nature de leurs liaisons à la surface (figure 1.5).

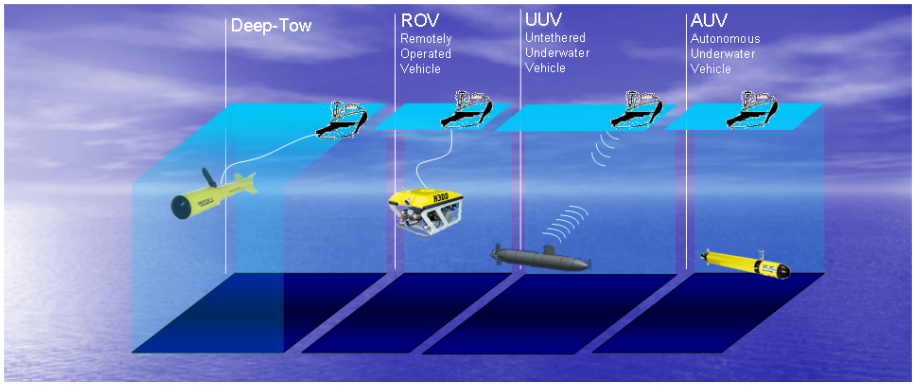


Figure 1.5 – Liaison à la surface des robots sous-marins

1.3.1 Contrainte de la liaison à la surface

Les poissons remorqués

Les poissons remorqués ou "Towed Fishes" sont des engins non motorisés pouvant transporter un certain nombre de capteurs. Ces engins sont remorqués depuis la surface grâce à un câble qui permet par la même occasion de remonter les données acquises. Ils sont principalement utilisés dans les opérations de cartographie, de reconnaissance du fond et pour des relevés physico-chimiques en profondeur. La simplicité relative de leurs mécaniques (pas d'organe articulé) leur permet de descendre facilement à de très grandes profondeurs.

Les ROV (Remotly Operated Vehicle)

Les ROV sont aussi reliés à la surface mais possèdent leurs propres moyens d'actionnement. Ils embarquent un ensemble de capteurs et les données acquises sont transmises via le câble de liaison à l'opérateur en surface. L'énergie nécessaire aux opérations du véhicule est convoyée depuis la surface par ce même câble. L'opérateur a également la possibilité d'envoyer à l'engin des commandes d'actionnement des moteurs pour pouvoir le déplacer et l'orienter selon les besoins de la mission. On parle alors de téléopération.

Le ROV est généralement conçu pour procéder à des interventions sur des structures immergées. Ils transportent un ou plusieurs bras manipulateurs (cf. figure 1.6), téléopérés en temps réel via un câble ombilical qui les relie à la surface. La présence de l'ombilical et du manipulateur génère des efforts de couplage perturbateurs qui doivent être explicitement considérés. Généralement, le ROV présente les propriétés suivantes :

- Iso-actionné : le système transporte autant de propulseurs que de degrés de liberté à contrôler. Les degrés de liberté restant sont naturellement stabilisés.

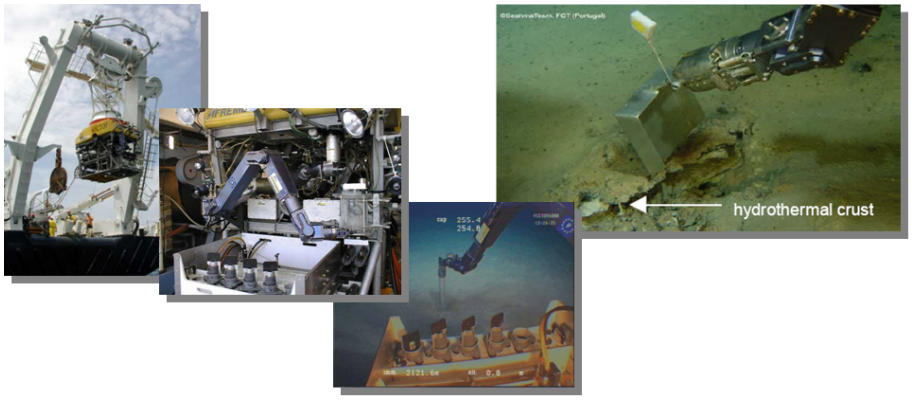


Figure 1.6 – Intervention sous-marine du ROV Victor 6000 de l'Ifremer [10]

- Vol stationnaire : ceci permet au système de se stabiliser autour d'une position désirée dans l'espace.
- Isotropie et holonomie : généralement de forme symétrique, ces systèmes sont conçus de façon à pouvoir exercer également des efforts dans toutes les directions.
- Câblé : le câble ombilical relie le ROV avec la surface. Ceci permet une téléopération en temps réel et le déport de la source d'énergie sur le bateau mère.
- Contrôle hybride position/force : la réalisation d'une intervention sous-marine sur une structure immergée implique de contrôler simultanément la position de la plate-forme et l'effort exercé par le manipulateur sur la structure.

Les UUV (Unmanned Underwater Vehicle)

Les UUV, engins sous-marins non habités embarquent leur propre énergie, leurs moyens d'actionnement et de perception. Une caractéristique importante de ces véhicules est leur degré d'autonomie. Ceux qui conservent un lien acoustique avec la surface leur permettant de recevoir de l'opérateur les ordres relatifs aux opérations qu'ils vont mener, sont appelés les UUV (Untethered Underwater Vehicle). Ils sont en partie autonomes mais l'opérateur peut prendre la main pour diriger le véhicule dans certaines situations d'urgence.

Ceux qui n'ont aucun lien avec la surface, les AUV (Autonomous Underwater Vehicle) reçoivent une mission avant d'être mis à l'eau et les prises de décision sur la manière dont ils doivent intervenir à chaque instant pour réaliser la mission sont à leur charge.

Ce type d'engin est conçu pour réaliser des missions au long-cours. Il est totalement autonome, ce qui implique une gestion fine de l'énergie embarquée et des capacités à prendre localement des décisions.

- Sous-actionné : le véhicule comporte moins d'actionneurs que de degrés de liberté. De plus, ces engins comportent une direction préférentielle de mouvement et des gouvernes permettent d'effectuer un changement de direction. Ceci implique que la capacité des actionneurs est dépendante de la vitesse d'évolution de l'engin dans le fluide.
- Gestion de l'énergie : ce type de véhicule transportant ses ressources énergétiques, il faut pouvoir minimiser sa consommation. Ceci implique de minimiser l'activité des actionneurs et de ne recruter les capteurs que lorsque leur mesure est nécessaire.

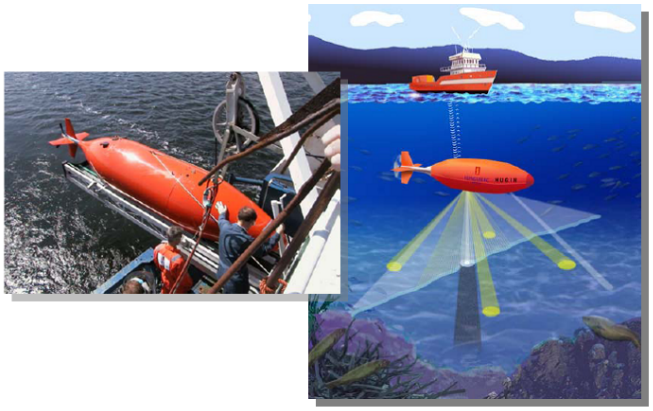


Figure 1.7 – Hugin 3000 opérant en mode UUV [11][12]

– Délais de transmission : l'absence de câble ombilical accroît le rayon d'action de ces engins, mais implique des difficultés pour la transmission d'information à la surface. La faible bande passante acoustique qu'offre le médium qu'est l'eau de mer implique de considérer des délais importants dans la transmission des informations.

Un certain nombre d'études comparatives [4] réalisées par des compagnies pétrolières et des développeurs d'AUV montrent, toujours un bénéfice pour l'utilisation des AUV.

1.3.2 Vers le déploiement de flottilles

Les flottilles de robots marins peuvent être considérées globalement comme des systèmes robotiques particuliers. En naviguant en formation, ils sont capables d'opérer sur des zones très étendues et d'effectuer des tâches irréalisables par une unité isolée (e.g. déplacement de structures). Leur composition peut soit être homogène c'est à dire constitué de robots de même structure matérielle et logicielle, soit hétérogène si les robots se différencient du point de vue matériel, logiciel ou les deux. La littérature présente des résultats théoriques sur la navigation en flottille mais les réalisations pratiques font défaut. Les problèmes encore non-résolus pour les véhicules seuls (e.g. problème d'interférence capteur), la lourde logistique nécessaire (plusieurs bateaux pour la mise à l'eau des différents robots), les faibles débits de communication et l'hétérogénéité des flottes justifient ces retards au niveau des expérimentations.

Le projet ASIMOV [13] est un exemple de flottille hétérogène : un véhicule de surface autonome (ASV) coopère avec un AUV en tant que relai de communication mobile (figure 1.8). L'ASV est équipé d'un GPS, un lien radio, d'un système de balises ultra courte et d'un lien acoustique. L'AUV transfère ses données de missions vers la surface et l'ASV envoie ses données GPS au fond.

1.3.3 Spécificité du milieu et impact sur la conception et le fonctionnement des engins sous-marins autonomes

Les systèmes marins autonomes font actuellement l'objet de recherches intensives, motivées non seulement par les enjeux considérables que représente la robotique marine,

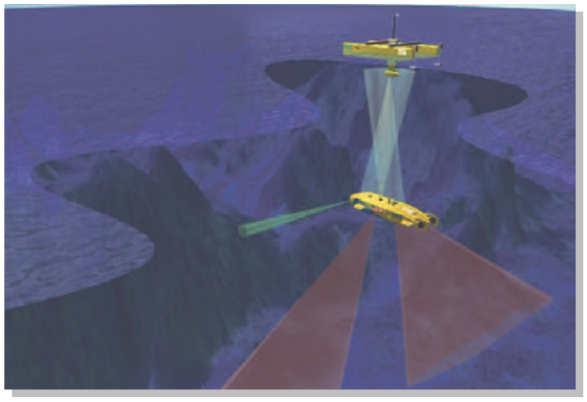


Figure 1.8 – Flottille hétérogène de deux robots : un ASV et un AUV.

mais aussi par les verrous scientifiques imposés par les contraintes du milieu marin. Les véhicules sous-marins doivent se mouvoir dans un environnement dense qui s'étend sur les trois dimensions de l'espace. Contrairement au milieu aérien ou spatial, les propriétés du milieu marin ne peuvent pas être négligées dans le cadre de l'étude des déplacements des engins même à vitesse faible. La conception de tels engins, et de leur commande, requiert la résolution de nombreux problèmes théoriques, en sus de l'expertise accumulée depuis des siècles dans le domaine de l'architecture navale et de la navigation maritime. Parmi les verrous actuels, nous en citons six, qui font actuellement l'objet de recherches intensives :

- L'architecture navale et la modélisation hydrodynamique représentent un passage obligé pour qui désire concevoir un engin marin performant. En effet la considération des phénomènes significatifs auxquels le système est confronté durant sa mission requiert une modélisation dynamique, dont la finesse influera fortement sur les performances globales du système contrôlé.
- Un autre problème fondamental est celui du positionnement, autrement appelé Navigation. Pour la robotique, le rôle du système de navigation est d'établir une estimation de l'état du système (positions et vitesses). Ceci implique de géo-référencer la situation du système et de le situer par rapport à l'objectif courant de la mission. Pour les véhicules de surface de type ASC (Autonomous Surface Craft), cette question est facilitée par l'utilisation de systèmes de type GPS. Par contre, dans le contexte sous-marin, le GPS n'étant plus disponible, de nouvelles stratégies doivent être mises en oeuvre.
- Le Guidage constitue la stratégie d'approche de l'objectif. Il est souvent négligé dans le cas de missions simples. Cependant, il revêt l'avantage de pouvoir fusionner différents critères de façon claire et peut se voir attribuer les tâches relatives à l'évitement de singularités ainsi que la combinaison d'un objectif global (suivi de chemin, de trajectoire,..) à la gestion de contraintes locales (évitement d'obstacles).
- Le contrôle a pour objectif de calculer les consignes à envoyer aux actionneurs de façon à garantir le suivi de la référence de guidage, telle que précédemment décrite.

En plus de ces problèmes de commande des véhicules liés aux caractéristiques mécaniques particulières de l'eau par rapport à l'air ou au vide, il faut ajouter la mauvaise transmission des ondes électromagnétiques. Ces ondes, support privilégié de l'information

dans l'air et dans l'espace, sont grandement atténuées par l'eau. Une très grande partie de l'instrumentation des robots terrestres, basée sur l'utilisation de ce type d'onde, ne peut être intégrée sur leurs homologues sous-marins. Notamment l'impossibilité d'utiliser un GPS pose le problème du positionnement géographique du véhicule partiellement résolu (pour de courtes distances) par une estimation du déplacement calculé à partir des accélérations subies.

Un autre type d'onde, l'onde sonore, se propage plus facilement dans les milieux de forte densité. Cette onde constituée d'énergie due aux vibrations du milieu parcouru constitue le meilleur moyen de transmission sous l'eau. Elle présente cependant des limitations notables :

- faible célérité, environ 1500 $m.s^{-1}$ (onde électromagnétique : 300 000 000 $m.s^{-1}$) ce qui introduit une certaine latence dans les communications.
- célérité variable, fonction des caractéristiques des couches d'eau traversée. Elle varie notamment en fonction de la profondeur ce qui doit être pris en compte lors de l'utilisation de l'instrumentation. En effet, certains capteurs basés sur le calcul du temps d'aller/retour d'une onde réfléchie (sur une surface dure) utilisent dans leurs calculs la vitesse du son dans l'eau pour déterminer la distance parcourue par l'onde.
- les pertes par divergences géométriques et par amortissement.
- les bruits du milieu marin (trafic maritime, perturbation sismique, organismes vivants etc.).
- interférences. Les engins sous-marins emploient souvent plusieurs capteurs acoustiques qui induisent des interférences s'ils fonctionnent à des fréquences proches ou harmoniques.

Sur ce type de véhicule autonome, l'énergie doit aussi être embarquée pour toute la durée de la mission. Les solutions à base de moteur thermique sont écartées dans le milieu sous-marin du fait de l'absence d'oxygène. L'utilisation de batterie réduit l'autonomie et exige une gestion de l'énergie au sein du véhicule.

1.3.4 Implication du besoin d'autonomie et de la nature des traitements sur le contrôle d'un AUV

Les AUV ont une grande autonomie décisionnelle puisqu'ils n'ont de contact avec l'opérateur qu'en surface, au début et à la fin de la mission. Cette autonomie dans un milieu fortement encombré, bruité et dont il est souvent impossible d'établir une carte à l'avance, se traduit par la nécessité de conférer au contrôleur du véhicule la capacité d'analyser le monde qui l'entoure et d'adapter son action (e.g. évitement d'obstacles) à son environnement tout en remplissant ses objectifs.

D'autre part le contrôleur de ce type de robot doit mettre en oeuvre un ensemble varié d'instruments de bord (centrale d'attitude, caméras, capteur de pression, sondeurs etc...) et de traitements informatiques (calculs de lois de commande, filtrage des données, prise de décision etc..) qui rendent sa conception complexe. Cette conception nécessite l'utilisation de techniques diverses (e.g. automatique, traitement du signal, traitement de l'image, informatique etc...) qui relèvent de corps de compétence différents ce qui accroît davantage sa complexité du fait de l'implication de nombreux acteurs dans le développement.

La construction d'un tel contrôleur nécessite l'étude au préalable de son architecture.

Cette architecture définit les différents éléments présents au sein du contrôleur ainsi que leurs relations.

Résumé du chapitre 1 :

- 71% de la planète sont constitués d'océans renfermant une immense richesse (carburant fossile, minerai, etc.) et présentant un enjeu au niveau industriel, scientifique et militaire.
- Les techniques d'exploitation de ces océans impliquant l'homme sont dangereuses et trop onéreuses laissant peu à peu la place à la robotique sous-marine.
- Nous pouvons classifier les robots sous-marins en quatres catégories : les poissons remorqués, les ROV (Remotly Operated Vehicle), les UUV (Untethered Underwater Vehicle) et les AUV (Autonomous Underwater Vehicle).
- Les AUV n'ayant aucun lien avec la surface durant leurs missions présentent des caractéristiques intéressantes (e.g. vastes zones d'exploration)
- L'autonomie dans le milieu marin impose aux AUV une certaine capacité décisionnelle pour la gestion des différentes contraintes.
- Le contrôleur des AUV, siège de la décision, doit être conçu selon une architecture bien précise.

Chapitre 2

Les architectures logicielles de contrôle

2.1 Introduction

Le besoin d'opérer de façon autonome dans des eaux peu ou très profondes, en réduisant les coûts au minimum (inconvénient des ROV-Romotely Operated Vehicle), amène les recherches à se concentrer sur l'élaboration de véhicules autonomes capables de se déplacer seuls et de mener à bien des tâches qui nécessitaient encore récemment l'assistance de l'opérateur humain.

Ce besoin d'autonomie dans un milieu en constante évolution requiert de la part du véhicule un certaine capacité à pouvoir, à chaque instant, évaluer son état et l'état de son environnement, le confronter avec la mission qu'il lui a été confiée et prendre des décisions cohérentes.

Les logiciels de navigation développés pour ces véhicules deviennent vite complexes tant au niveau de leur conception et de leur implémentation qu'au niveau de leur analyse. En effet un minimum (à défaut d'un maximum) de propriétés doivent être garanties avant de mettre à l'eau un AUV au risque sinon de le perdre. Un logiciel de navigation peut se subdiviser en deux grandes parties, l'une s'occupant de tout ce qui a trait au raisonnement logique (planification, choix de la meilleure solution) et l'autre gérant les capteurs, les actionneurs et la commande. Dans la littérature, il est souvent fait référence aux critères de qualité de conception d'un logiciel de navigation ; D.B. Marco, A.J. Healey et R.B. McGhee [14] proposent d'évaluer une architecture logicielle selon huit points :

1. Le contrôleur permet-il une évaluation facile de la réponse du système et est-il possible de régler simplement les paramètres de commande ?
2. Cela peut-il être fait en temps réel durant un test ?
3. Un nouveau capteur peut-il être ajouté sans nécessiter de grandes modifications au niveau du logiciel ?
4. Dans quels niveaux du code faut-il intervenir et combien de fonctions nécessitent un changement lors de l'ajout d'un nouveau capteur ?
5. Quelle quantité de code doit-on changer si l'on veut ajouter ou retirer une phase dans la mission ?

6. Comment le code doit-il être modifié pour évaluer les performances d'un capteur ou d'un actionneur ?

7. Est-il facile de changer pour accepter les données fournies par un autre ensemble de capteurs ?

8. Est-il facile de modifier les conditions qui définissent les signaux de transition ?

Ces huit points reposent en fait sur un ensemble de critères plus large. En effet, on peut mettre en évidence plusieurs critères sur lesquels doit se baser la conception d'architectures logicielles de contrôle des robots sous-marin autonomes :

- **Modularité** : la complexité d'un logiciel de navigation impose au concepteur de le diviser en composants plus petits qui peuvent être développés, implémentés et testés séparément.
- **Ouverture** : le logiciel doit pouvoir être développé progressivement, de nouveaux composants logiciels doivent pouvoir être ajoutés par différents acteurs, tout en réutilisant les composants existants.
- **Flexibilité** : le logiciel doit présenter une structure flexible permettant d'ajouter ou de retirer facilement des composants logiciels durant ou après la phase de développement.
- **Temps réel** : un signal de contrôle doit pouvoir être généré dans n'importe quelle circonstance et dans un temps déterminé.
- **Gestion des contraintes** : le véhicule doit pouvoir atteindre les objectifs à long terme de la mission tout en répondant à des contraintes immédiates.
- **Fiabilité et robustesse** : le véhicule doit pouvoir exécuter une séquence d'objectifs planifiée en présence d'incertitudes et de gérer les situations dont l'occurrence est temporellement imprévisible (obstacle, entrée d'eau...).
- **Surveillance du bon fonctionnement** : le véhicule doit pouvoir vérifier le bon fonctionnement de toute l'instrumentation, détecter et réagir de façon adéquate à une quelconque défaillance.

Certains de ces critères sont depuis fort longtemps considérés dans la conception d'applications informatiques au sens "applications générales" (i.e. non orientées contrôle). Les avancées du génie logiciel n'ont cependant été considérées dans le contexte du contrôle que tardivement.

En effet, leur exploitation dans notre contexte "robotique" est apparu dans la littérature vers les années 90 [15]. Aujourd'hui, les travaux "pluridisciplinaires" (robotique/informatique) sur le sujet sont conséquents tant au niveau international [16] qu'au niveau national [17] [18]. Ainsi des propositions issues des approches à composants logiciels et de l'ingénierie des modèles, pour ne citer que ces tendances, sont d'actualité [19].

Nos travaux, centrés sur la gestion contextuelle de tâches dans le cadre du contrôle, n'adresseront pas ces problématiques de méthodologie, de génération automatique de code, etc... Nous nous attacherons cependant à prendre en compte les critères énoncés pour favoriser l'intégration de nos propositions dans ce type d'approche.

2.2 Principes d'organisation

Un robot mobile sonde son environnement puis agit afin de se diriger vers son objectif. Les capteurs et les actionneurs sont les organes physiques qui vont permettre respectivement d'évaluer l'état proprioceptif et extéroceptif du système et d'effectuer une action (mouvement). Entre les données générées par les capteurs et les commandes envoyées aux actionneurs se trouve le logiciel de contrôle. C'est au sein de ce dernier que se construit la décision. Cette décision qui va affecter l'état du robot par rapport à son environnement, doit avoir pour finalité l'objectif (figure 2.1) spécifié par l'utilisateur et doit tenir compte du contexte (situation du robot, dangers éventuels,...). Plusieurs types de traitements interviennent au sein du logiciel (gestion des capteurs, filtrage des données, planification, calcul de loi de commande, etc...). La manière dont ces traitements sont implémentés (éléments logiciels) ainsi que les liens de communication et de contrôle qui les relient sont représentés dans l'**architecture** du logiciel. L'élaboration d'un logiciel, et à plus forte raison d'un logiciel complexe tel un contrôleur robotique, commence par la conception de son architecture : entre autres doivent être définies les entités (composants) constitutives et leur structuration donnant naissance à l'architecture.

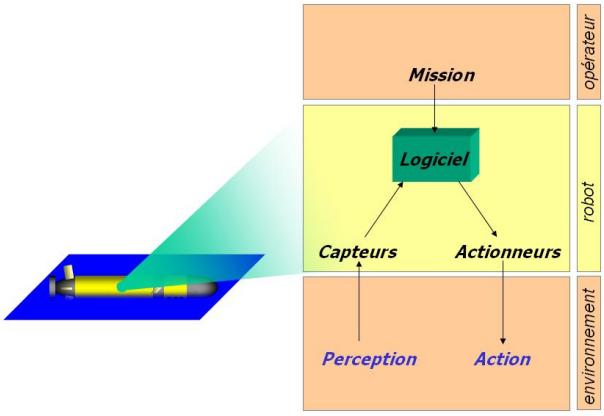


Figure 2.1 – Logiciel de commande

Les approches de structuration de ces architectures logicielles peuvent être classifiées selon trois catégories principales : les architectures réactives (ou comportementales), les architectures hiérarchisées et les architectures hybrides. Les deux premières mettent l'accent respectivement sur la rapidité de réaction et sur la clarté de la structuration en terme de "niveau" de compétence.

Au regard des inconvénients de ces deux types d'approches, la tendance est aux architectures dites hybrides au sens où elles intègrent le besoin de structuration en niveaux tout en préservant la réactivité nécessaire. Nous allons illustrer plus en détails ces différents types d'architectures.

2.2.1 Architectures réactives

Dans une architecture réactive plusieurs modules relient les entrées capteurs aux actionneurs (figure 2.2). Chaque module implémente un comportement, c'est à dire une

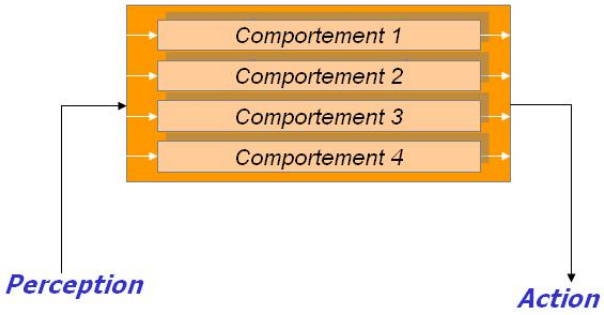


Figure 2.2 – Architectures réactives

fonctionnalité élémentaire du robot associant à chaque vecteur d'entrée (ensemble des valeurs capteurs) un vecteur de sortie appliqué aux actionneurs. Ces comportements sont dit "réactifs" car ils fournissent immédiatement une valeur de sortie dès qu'une valeur se présente en entrée. Par exemple un comportement dédié à l'évitement d'obstacle dans un robot mobile, appliquera une commande aux actionneurs des roues de façon à faire changer la direction à chaque fois que le capteur de proximité indique la présence d'un objet.

Inspirées de l'observation des comportements animaux, ces architectures sont construites selon l'idée que de la composition d'un ensemble de comportements élémentaires simples peut émerger un comportement plus évolué. L'intégration de plusieurs comportements amène souvent des contradictions au niveau des commandes générées par ces derniers. On résout alors ce problème en ajoutant un module d'arbitrage qui va se charger de pondérer les sorties de chacun des comportements pour générer une sortie unique (figure 2.3).

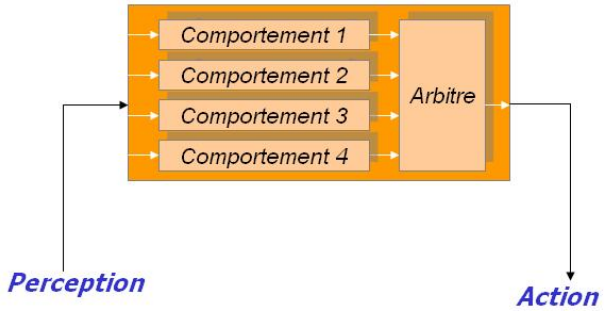


Figure 2.3 – Architectures réactives arbitrées

Le comportement final (global) du robot est alors étroitement lié à cette pondération. Selon l'application que l'on veut faire réaliser au robot, il faut alors déterminer un ensemble de "poids" qui vont être utilisés par le module d'arbitrage. Toute la complexité réside dans la recherche de la bonne combinaison de valeurs numériques (poids) en intégrant les objectifs de haut-niveau, qui va donner le comportement global souhaité.

Pour palier à cette difficulté, vers la fin des années 80, Brooks a proposé l'architecture dite "subsumption" [20]. Selon cette approche, les différents comportements sont classés par niveaux de compétence. Chaque niveau renferme un ensemble de modules qui implémentent un comportement donné. Chaque module d'un niveau peut inhiber les entrées

ou les sorties de modules du niveau immédiatement inférieur (figure 2.4).

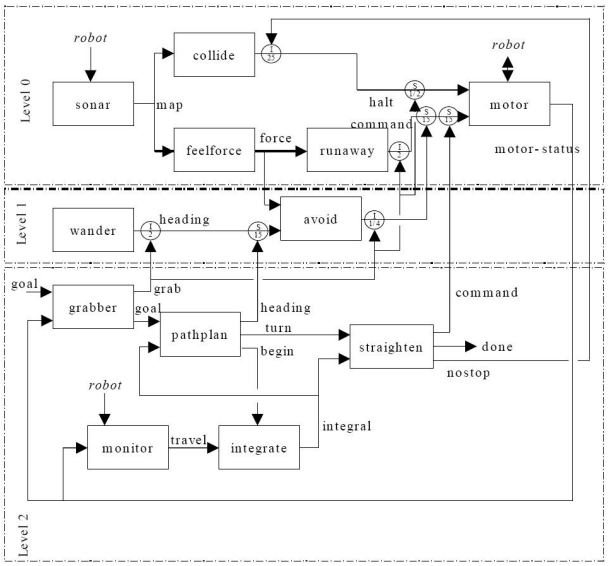


Figure 2.4 – Architecture subsumption [20]

Une autre variante des travaux de Brooks est l'architecture DAMN proposée par Rosenblat [21]. Dans cette architecture (figure 2.5) plusieurs comportements élémentaires tels que le suivi de route ou l'évitement d'obstacle adressent des votes à un module spécial appelé "arbitre de commande". Ces entrées sont combinées et le résultat est une commande qui sera transmise au contrôleur du véhicule. Un poids est assigné à chaque comportement reflétant sa priorité relative dans le contrôle du véhicule. A la différence de l'architecture subsumption, les poids peuvent varier au cours du temps. En effet, un module nommé "générateur de mode" va calculer ces derniers tout au long de la mission et favoriser ainsi à chaque instant certains comportements par rapport à d'autres, selon la situation rencontrée.

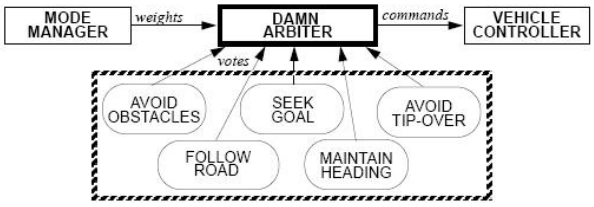


Figure 2.5 – Architecture DAMN [21]

Plusieurs expériences ont montré l'efficacité de ces architectures en robotique mobile. La réactivité les caractérisant a mené à des résultats souvent impressionnants dans la navigation en milieu encombré. Cependant la complexité de conception de ces architectures ne permet pas de les utiliser pour des applications demandant un comportement plus évolué que la navigation. L'objectif global du robot étant implémenté sous forme d'un réseau

de modules, il n'est pas possible d'enchaîner plusieurs objectifs ou de changer facilement la mission initiale.

2.2.2 Architectures hiérarchisées

Le modèle d'organisation des architectures **hiérarchisées**, également appelées architectures délibératives (cf. fig. 2.6), centre la conception sur le système décisionnel. Ces architectures sont organisées, dans la plupart des propositions, en plusieurs couches (également appelées niveaux) hiérarchisées [22] [23] [24]. Une couche communique uniquement avec la couche directement inférieure et la couche directement supérieure. Ces architectures comportent typiquement trois couches :

- La couche *fonctionnelle* contient des modules de perception qui sont en charge de la transformation des données numériques, provenant des capteurs, en données symboliques. Elle contient également des modules de génération de trajectoires et des modules d'asservissement, qui sont responsables de la transformation des données provenant du niveau supérieur, en données numériques applicables aux actionneurs.
- La couche *exécutive* est en charge de la supervision des tâches du robot (e.g. "se diriger vers la source de chaleur la plus proche"). Elle contient pour cela des modules qui dirigent l'exécution des modules de la couche *fonctionnelle*.
- La couche *décisionnelle* est en charge des mécanismes de planification (i.e. création et supervision de plans). Elle gère la façon dont le robot va enchaîner différentes tâches afin de réaliser chaque objectif de sa mission. Les mécanismes de décision de plus haut niveau, appelés traditionnellement mécanismes de *délibération* sont les mécanismes centraux dans ces architectures [25], basés sur des techniques d'intelligence artificielle.

L'information provenant des capteurs est propagée verticalement, de la couche fonctionnelle vers la couche décisionnelle, en traversant potentiellement toutes les couches intermédiaires. L'information est transformée, au fur et à mesure des traitements réalisés dans chaque couche, en une information de plus en plus abstraite. Ceci nécessite des traitements adéquats afin de fusionner les données et d'en extraire les informations plus abstraites nécessaires au plus haut niveau. Inversement, la propagation de la décision (sous forme de données symboliques) se fait de la couche décisionnelle vers la couche fonctionnelle, en traversant successivement toutes les couches intermédiaires. La couche fonctionnelle applique alors, en fonction de cette décision, les asservissements adéquats (i.e. activation des modules d'asservissement qu'elle contient).

Le principal problème des contrôleurs conçus avec de telles architectures, vient de leur manque de réactivité, c'est-à-dire la capacité de réagir en temps voulu à des modifications dans l'environnement. Ce manque de réactivité provient de l'organisation en couches : le transfert de l'information doit obligatoirement se faire en traversant toutes les couches intermédiaires. Ainsi, une adaptation décidée dans la couche décisionnelle (ex : modification du plan courant) se base sur des informations provenant exclusivement de la couche exécutive, qui elles-mêmes proviennent du traitement d'informations provenant de la couche fonctionnelle. La décision d'adaptation est alors propagée vers la couche fonctionnelle en traversant les couches intermédiaires. Avec cette forme d'organisation, nous constatons que les traitements réalisés dans les couches intermédiaires induisent des latences d'autant plus importantes que le nombre de couches (et donc de traitements) est grand. Nous

constatons également que dans de nombreux cas, traiter l'adaptation dans la couche décisionnelle se révèle très coûteux en temps de réaction, en particulier à cause de la lourdeur des mécanismes de délibération. En effet, l'occurrence d'événements temporellement imprévisibles induit souvent une replanification au niveau de la couche décisionnelle. Si le calcul de la planification est sans garantie de temps borné (en adéquation avec la dynamique du robot), le robot doit alors être "arrêté". Dans les architectures délibératives, le comportement global du robot est défini en fonction de hiérarchie de mécanismes de décision/contrôle : ce comportement est donc maîtrisé et compréhensible par l'homme, mais l'approche manque globalement de réactivité.

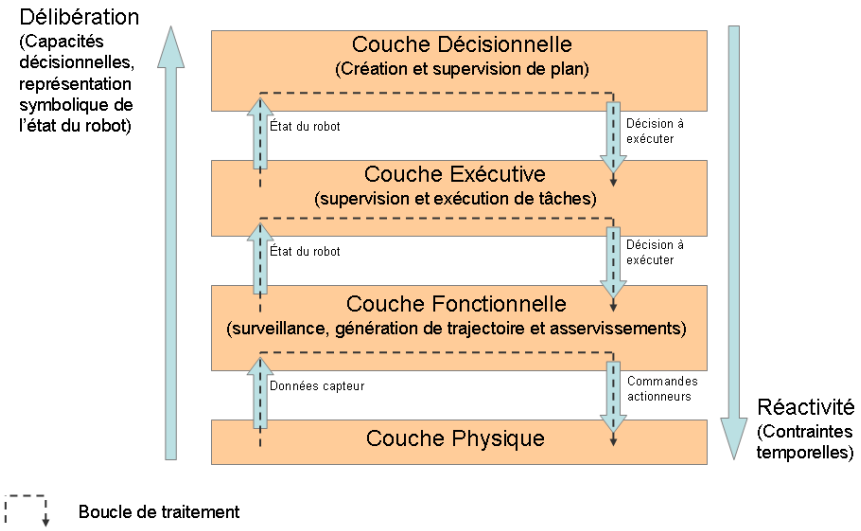


Figure 2.6 – Modèle d'organisation des architectures délibératives

2.2.3 Architectures mixtes ou hybrides

Les architectures mixtes permettent de prendre en compte à la fois l'aspect décisionnel et réactif. Ces architectures se développent pour répondre à la nécessité qu'ont les robots actuels de fournir un comportement autonome dans un environnement mal connu avec lequel le robot doit interagir. Ces architectures intègrent une hiérarchisation des couches permettant la symbolisation et donc la prise de décision. A cette hiérarchisation s'ajoutent des boucles réactives imbriquées permettant à chaque couche de fournir des réactions adaptées à sa dynamique.

2.3 Architectures en robotique

2.3.1 Architecture IDEA

L'architecture IDEA (Intelligent Distributed Execution Architecture) résultant des expérimentations réalisées par la NASA sur la sonde DS1, repose sur une approche multi-agent. Chaque agent peut être indifféremment un module fonctionnel, un planificateur, un système de diagnostic...

Chaque agent est alors construit autour d'un couple planificateur/exécuteur et d'un modèle de contraintes et de compatibilités qui spécifie les évolutions possibles des différentes variables d'état. La réactivité est alors distribuée sur chacun des agents qui prennent en charge chacun des événements qui leurs sont propres (détection, traitement, réaction) [26].

Le fonctionnement des agents est basé sur les "procedure" et les "token". Un "token" est une unité de temps durant laquelle un agent peut exécuter une "procedure". Une "procedure" se présente sous la forme $P(i_1, ..., i_n \rightarrow m_1, ..., m_k \rightarrow o_1, ..., o_m; s)$ où les i_i, m_i et o_i représentent respectivement un argument d'entrée (input), de mode et de sortie (output), et une valeur de retour s (status). Une "procedure" s'exécute jusqu'a sa terminaison ou jusqu'a être interrompue par l'agent (fin du "token" alloué à la "procedure").

Les agents communiquent entre eux par échange de messages. Ces messages peuvent être des requêtes de lancement de "procedure" au sein de l'entité réceptrice ou être porteur d'un but que l'agent cible doit atteindre et sont tous deux traités sous forme de "token". Les arguments, la date de début, la date de fin de chaque "token" reçu sont pris en compte par l'agent pour déterminer le plan à exécuter. Un agent IDEA peut communiquer avec plusieurs autres agents. Ces communications sont les supports des flux de contrôle et ne sont pas contraintes : deux agents peuvent se contrôler l'un l'autre.

Un agent exécute les "token" présents dans sa base de données de plan (figure 2.7). Ces "token" étant issus des buts transmis par un agent tiers ou suite à une planification interne ayant généré des sous-buts. Cette base de données est partitionnée en lignes de temps (timelines) représentant chacune l'évolution temporelle d'une propriété d'un sous-système (e.g. caméra). Les "token" d'une même ligne de temps sont exécutés séquentiellement, ceux de deux lignes différentes sont exécutés en parallèle.

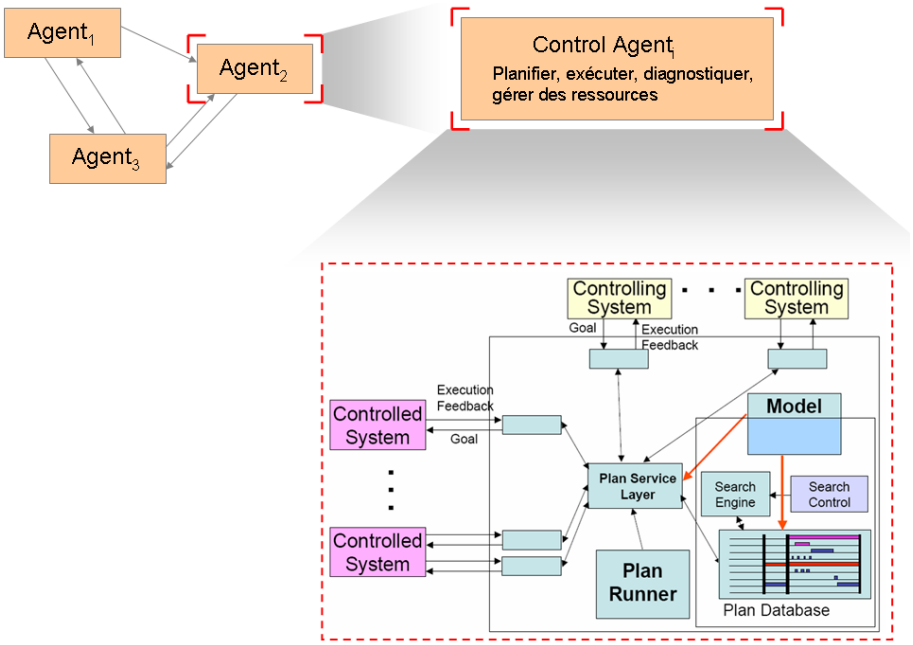


Figure 2.7 – Architecture IDEA

2.3.2 Architecture du LAAS

Le LAAS (Laboratoire d'Analyse et d'Architecture des Systèmes, Toulouse, France) propose un environnement logiciel constitué d'outils pour modéliser, programmer, exécuter et dans une certaine mesure valider, des architectures de contrôle [27]. Cet environnement s'appuie sur un modèle (cf. fig. 2.8) qui définit un découpage d'une architecture de contrôle en trois couches (appelées niveaux dans cette proposition) :

- Le niveau *fonctionnel* constitue l'interface entre les entités des couches supérieures et la partie physique du système. Il est le siège des fonctions de base du système : fonctions sensori-motrices, fonctions d'asservissement (e.g. navigation), fonctions de traitement (e.g. planificateur de trajectoire, segmentation d'image). Toute fonction est encapsulée dans un *module* (non donné aux entités logicielles constituant la couche fonctionnelle), généré par l'outil GenoM [28]. Un module offre un ensemble de services accessibles via des requêtes asynchrones (permettant de le démarrer/arrêter/paramétrer), chaque service offrant un *rapport* de fin d'exécution (i.e. bilan d'exécution). Un module est en charge d'une ressource physique ou logique et dispose des fonctions nécessaires au contrôle de cette ressource, ainsi qu'un contexte d'exécution propre.
- Le niveau de *contrôle d'exécution*, est en charge de vérifier les requêtes envoyées aux modules du niveau fonctionnel et l'utilisation des ressources du robot. Il agit comme un filtre qui peut refuser certaines requêtes (provenant de l'opérateur ou d'un module) en fonction du contexte du robot (actions en cours, ressources utilisées, etc.) et d'un modèle défini par l'opérateur. Ce niveau sert principalement à assurer certaines qualités de robustesse et de sûreté de fonctionnement au contrôleur.
- Le niveau *décisionnel* contient les mécanismes de décision du robot, en particulier la production et la supervision de plans et la réaction à des situations particulières (e.g. pannes matérielles). Ce niveau comprend deux entités : un *exécutif procédural* et un *planificateur/exécutif temporel*. Le planificateur/exécutif temporel gère la planification de la mission globale du robot (production de plans pour réaliser les missions) sur un horizon à long terme. Il peut également replanifier une mission en prenant en compte l'ajout et la suppression dynamique de buts (par l'opérateur) et les échecs d'exécution (e.g. time-out ou échec d'une des actions définies dans le plan). L'exécutif procédural est responsable de la supervision des missions envoyées par les opérateurs humains. Pour cela il interagit avec le planificateur/exécutif temporel afin que celui-ci génère le plan permettant de réaliser la mission reçue. Une fois ce plan généré, l'exécutif procédural est chargé d'exécuter les actions définies dans le plan de mission. Pour cela il interagit avec les modules du niveau fonctionnel via le niveau de contrôle d'exécution (qui filtre ses requêtes et les bilans d'exécution des modules fonctionnels). A la fin de la réalisation d'une action (terminaison d'un module fonctionnel), il envoie au planificateur/exécutif temporel un bilan notifiant un succès ou un échec de l'action courante.

Le modèle d'organisation des architectures de contrôle proposé par le LAAS est basé sur les principes d'organisation des architectures délibératives hiérarchisées. En effet, les mécanismes de délibération (planification et supervision de plans) du niveau décisionnel y sont centraux. A l'image des dernières architectures délibératives hiérarchisées telle que 4D/RCS [24], il existe des mécanismes d'adaptation à différents endroits : dans la

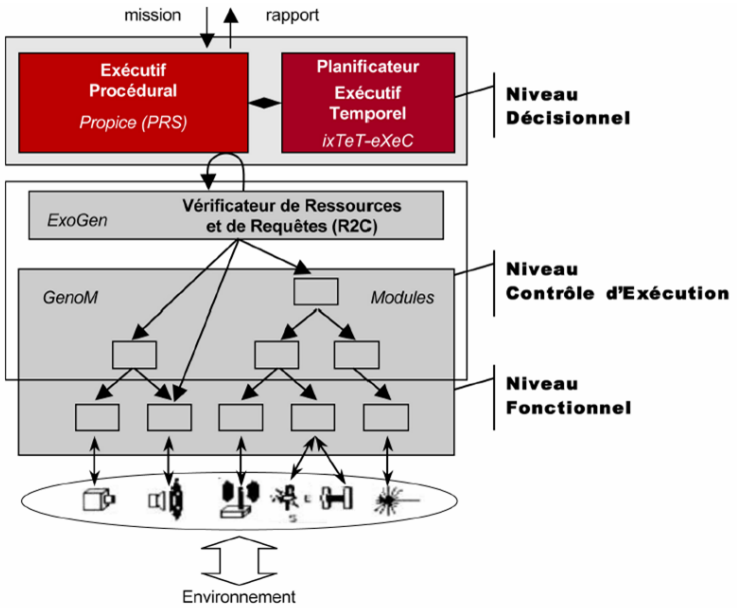


Figure 2.8 – Modèle d'organisation de l'architecture de contrôle du LAAS [27]

hiérarchie des modules fonctionnels (adaptation des modules fonctionnels utilisés par un module fonctionnel plus global), dans l'exécutif procédural (adaptation, dans une certaine mesure, des modules fonctionnels utilisés pour réaliser une action) et dans le planificateur exécutif procédural (reformulation sur échec d'action). La décision d'adaptation du comportement du robot peut être prise aux plus hauts niveaux décisionnels (planification), mais peuvent également résulter d'actions réflexes programmées dans des modules de la couche *fonctionnelle*. L'autre constat est que l'information ne fait pas de saut entre les couches, par exemple directement de la couche fonctionnelle vers la couche décisionnelle, elle passe automatiquement par la couche de contrôle d'exécution. On remarque néanmoins que l'interaction entre la couche décisionnelle et la couche fonctionnelle est quasi-directe, la couche de contrôle d'exécution n'existant que pour assurer sûreté et robustesse du contrôle. On pourrait donc voir, in fine, ce modèle d'organisation comme étant à deux couches. Tout ceci rend difficile le classement de ce modèle d'organisation dans la catégorie des architectures délibératives, ou dans celle des architectures mixtes.

L'organisation d'une architecture développée suivant l'approche du LAAS est relativement rigide puisqu'elle décrit les couches d'une architecture de contrôle de façon monolithique. En effet, si l'utilité de chaque couche est établie, les différents types d'entités utilisées au sein de la couche fonctionnelle ne sont pas explicitement identifiés. Les types d'entités des couches décisionnelle et exécutive sont pré-fixées, et le modèle est fourni avec un ensemble de composants prédéfinis pour chacun de ces types, l'utilisateur n'ayant qu'à les configurer. Nous pensons que cette vision très agrégée est un frein pour l'identification des différentes entités. Une vision à grain plus fin des différents niveaux, favoriserait l'identification et la la réutilisation de leurs entités constitutives. Ces propos doivent être nuancés en prenant en compte le fait que la couche fonctionnelle peut être organisée en

une hiérarchie de modules en interaction. Identifier les différents types de modules (e.g. asservissement, perception, navigation, etc.) et leurs interactions typiques permettrait de mieux comprendre la façon dont le contrôle est organisé.

Le modèle d'architecture est adapté au développement de systèmes robustes, capables de poursuivre leur mission après des pannes partielles ou des échecs. Il est supporté par un outillage très complet et spécialisé, facilitant la mise en œuvre et la validation de certains aspects (mais qui d'autre part, limite la modularité).

- Le planificateur/exécutif temporel IXTET-EXEC permet de générer et exécuter des plans temporels ; il est capable de prendre en compte dynamiquement de nouveaux objectifs de missions et les échecs d'exécution. Il est basé sur des techniques de résolution de problèmes de contraintes (CSP).
- L'exécutif procédural PRS/Propice permet de superviser la réalisation d'actions ; il est réactif aux événements provenant de l'opérateur ou de la couche inférieure.
- Le contrôleur d'exécution R2C reconstitue dynamiquement l'état des ressources du robot (en fonction des requêtes et bilans d'exécution qu'il contrôle) ; il permet de filtrer les requêtes en fonction de cet état et d'un modèle formel des états acceptés ou non.
- L'outil Genom [28] permet de générer les modules du niveau fonctionnel à partir d'une description formelle (basée sur des machines à état finis) du comportement des modules.

2.3.3 CLARATY

CLARATy (Coupled Layer ARchitecture for Robotic Autonomy) [29] [30] est un modèle d'architecture mixte proposé par le NASA Jet Propulsion Laboratory et l'université Canergie Mellon de Pittsburg. CLARATy est vue comme la proposition du futur pour le développement de contrôleurs de robots à la NASA. Dans le modèle CLARATy, la décomposition d'une architecture de contrôle se fait en deux couches : la couche fonctionnelle et la couche décisionnelle.

La description de la couche fonctionnelle repose sur une approche objet, qui permet une forte réutilisabilité du code et l'extension facilitée des fonctionnalités d'un contrôleur. L'approche objet permet, à travers une décomposition modulaire et hiérarchique, de représenter un système robotique à différents niveaux d'abstraction. Les classes abstraites servent à représenter des concepts génériques, comme par exemple le concept de *locomotor* qui encapsule les fonctions de locomotion d'un robot. Ces classes peuvent être alors spécialisées en fonction du matériel physique utilisé (roues, capteurs, bras mécanique etc.). Cela favorise l'optimisation des traitements en fonction de ce matériel et cela permet, dans une certaine mesure, de découpler les procédures génériques des capacités spécifiques d'un système. Trois catégories de classes d'objets sont définies dans CLARATy :

- les *Data Structure Classes* permettent de représenter, manipuler et stocker les données sous une forme standard, base nécessaire à une réutilisation efficace. Ces classes peuvent être génériques ou spécifiques à un domaine d'application. Les bits, les vecteurs, les matrices, les images, les messages, les quaternions, sont des exemples de tels objets.

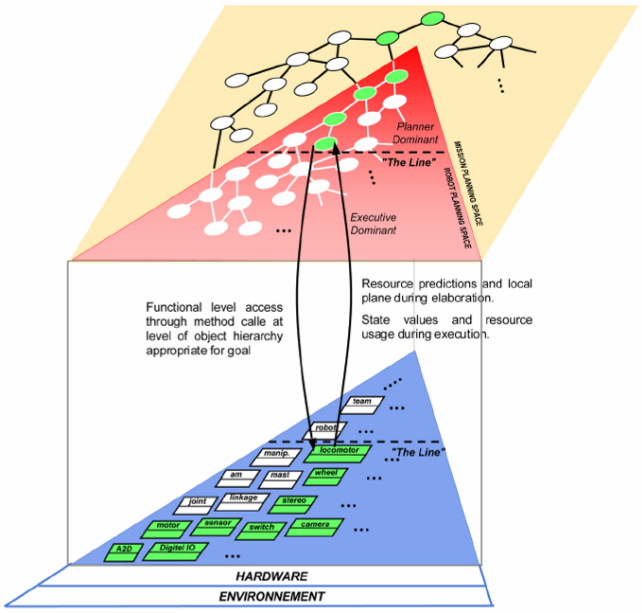


Figure 2.9 – Modèle d'organisation de l'architecture CLARATy [31]

- les *Generic Physical Classes* permettent de représenter des entités physiques, par exemple le locomoteur, la caméra, la roue, le manipulateur. Chacune de ces classes possède une interface qui définit les fonctionnalités de l'élément matériel physique correspondant. Par exemple, l'interface de la classe locomoteur contient des opérations permettant de déplacer le véhicule en spécifiant le chemin à suivre et sa vitesse. La spécialisation de ces classes permet classiquement de leur ajouter des fonctionnalités, d'autre part elle permet d'adapter les concepts physiques abstraits à un système robotique particulier, permettant ainsi de gérer spécifiquement le matériel utilisés dans le robot (les auteurs parlent alors de *Specialized Physical Classes*).
- les *Generic Functional Classes* sont des classes qui représentent un algorithme générique. Une telle classe utilise, pour définir son contenu, un ensemble de *Generic Physical Classes*. Par exemple, la classe navigateur permet de contrôler le déplacement d'un robot dans une zone (évaluation du terrain, calcul du chemin, sélection des actions à entreprendre, etc.). La description détaillée du navigateur peut être trouvée dans [32]. Ces classes peuvent être spécialisées afin d'adapter un algorithme à un système robotique particulier, ce qui sert essentiellement à optimiser des algorithmes en fonction du matériel considéré.

La couche décisionnelle est en charge de la gestion de l'"intelligence" du robot ; elle est spécifique à chaque robot. Elle est utile à la planification et à l'exécution de plans. Elle permet de transformer les objectifs de haut niveau d'abstraction, reçus par le robot, en une série d'objectifs ordonnés suivant un ensemble de contraintes connues et selon l'état du système. Afin de réaliser des objectifs, la couche décisionnelle interagit avec les objets de la couche fonctionnelle, à travers un protocole client/serveur lui permettant : de connaître l'état des ressources physiques du robot à un moment donné, de contrôler

les asservissements du robot, de demander l'exécution de traitements spécifiques (e.g. calcul de trajectoire). Cette couche est décrite suivant un *Goal Net*, qui représente la décomposition des buts de haut niveau d'abstraction en sous-buts. Chaque but ou sous-but peut être décomposé jusqu'à arriver aux buts feuilles qui accèdent directement à la couche fonctionnelle. Cette décomposition du niveau décisionnel sous forme d'un *Goal Net* reste conceptuelle, différents outils pouvant servir à sa mise en œuvre.

L'utilisation de l'approche objet permet aux auteurs de CLARATy de définir un ensemble de Frameworks orientés objets dédiés à la construction d'applications robotiques (par exemple, un framework de *Generic Physical Classes*[31]). CLARATy a pour principal atout la réutilisation des classes de la couche fonctionnelle, les auteurs annonçant un taux de réutilisation du code de l'ordre de 80% à travers les différents projets de la NASA utilisant CLARATy. L'apport de CLARATy vient de la possibilité de représenter explicitement les éléments matériels constituant le robot, et les fonctionnalités qui y sont rattachées, sous forme d'objets. L'approche objet est une solution intuitive pour représenter la partie physique d'un robot. Elle permet de décrire ou programmer le contrôle d'un robot à différents niveaux d'abstraction, du plus générique au plus spécialisé en fonction du matériel. Elle permet la mise en œuvre progressive de Frameworks spécialisés (locomotion, manipulation, vision, etc.) assurant une très forte réutilisation. D'un autre côté, la description du contenu du niveau décisionnel demeure, à notre connaissance, relativement "floue", les auteurs n'imposant pas de formalisme ou d'outil spécifiques. La proposition reste néanmoins intéressante, car les auteurs suggèrent que la couche décisionnelle peut elle-même être décomposée en sous-mécanismes de prise de décision, suivant une organisation "récursive". En conclusion, le modèle d'organisation de CLARATy peut être qualifié de mixte, puisqu'il décompose les architectures en deux couches en interaction directe : l'information transite directement d'une couche à l'autre et n'a donc pas plusieurs couches intermédiaires à franchir, ce qui permet d'optimiser la réactivité. Notons, pour conclure, que, tout comme l'approche du LAAS (présentée précédemment), CLARATy reste essentiellement inspirée des architectures délibératives classiques.

2.3.4 Architecture de l'Institut Supérieur Technique de Lisbonne (ISTL)

Les travaux de l'ISTL sont basés sur le développement d'une architecture de contrôle pour un robot sous-marin (AUV) dénommé MARIUS [33][34]. Ils proposent une architecture hiérarchisée suivant trois couches. La couche la plus basse dénommée "System tasks" met en oeuvre des machines synchrones ayant en charge les asservissements du robot. Cette couche fonctionnelle permet de créer des primitives ("vehicle primitives") qui sont équivalentes à une spécification paramétrée d'opérations élémentaires. Ces primitives mettent en oeuvre et coordonnent l'exécution d'un ensemble de tâches parallèles.

L'ensemble des "vehicles primitives" forme une bibliothèque permettant d'élaborer des "Mission procedures". Ces procédures sont des enchaînements logiques et temporels de primitives. Le niveau mission est réactif par rapport à un ensemble d'événements attachés à l'environnement. Les primitives remplissent des fonctions, parfois propres aux véhicules sous-marins, comme la navigation, le positionnement, la communication, la détection d'amers ou d'obstacles dans l'environnement. Le modèle hiérarchique a ici une place plus prépondérante. Cependant, au niveau décisionnel, le niveau "mission proce-

dure" possède une réactivité qui lui permet d'effectuer des adaptations de la séquence fournie par le "mission program". Cette réactivité est en fait très limitée et repose sur un ensemble d'alternatives dans le déroulement du programme de la mission. On fournit un plan qualitatif intégrant la possibilité d'occurrence d'un ensemble d'événements et on obtient par la suite un plan a posteriori représentatif du chemin que l'on a parcouru au sein des alternatives.

Résumé du chapitre 2 :

- Trois différents types d'architecture : comportementales, délibératives et mixtes.
- Les recherches s'orientent de plus en plus vers les architectures mixtes.
- Les architectures mixtes sont généralement constituées de trois niveaux hiérarchiques : niveau délibératif (en charge des mécanismes de planification), niveau exécutif (en charge de la supervision des tâches du robot) et le niveau fonctionnel (en relation avec l'instrumentation).

Chapitre 3

Ordonnancement de tâches

3.1 Introduction

Au chapitre 1 nous avons vu que les contrôleurs des véhicules sous-marins autonomes doivent intégrer des contraintes temporelles liées à leur commande (respect des dates d'échantillonnage) ainsi qu'à l'utilisation de leur instrumentation (gestion des interférences capteur). Répondre à ces contraintes temporelles nécessite de maîtriser les activités (sur une échelle de temps) des différents traitements qui ont lieu au sein du contrôleur. Les architectures de la littérature présentées au chapitre 2 ne permettent pas d'exprimer et de prendre en compte facilement ces contraintes temporelles.

Afin de répondre de façon satisfaisante aux différentes contraintes de temps imposées par le système, il est nécessaire de pouvoir proposer un ordre d'exécution des différents traitements. Cet ordre relève d'un ordonnancement dont nous étudierons les grandes lignes dans ce chapitre dans le cas qui nous intéresse : utilisation d'un seul processeur.

3.2 Eléments d'ordonnancement

3.2.1 Définitions et notions générales

Les applications temps réel sont des applications pour lesquelles la principale contrainte à respecter concerne le temps. Aujourd'hui les applications temps réel sont rencontrées dans de nombreux domaines stratégiques tels que la commande des procédés industriels, la robotique (mobile, télé robotique), les systèmes embarqués, les domaines des transports (automobile, aérospatial, navale), surveillance des centrales nucléaires, le traitement du signal et de l'image, etc.

Dans ce qui suit, nous nous intéressons à la notion importante de l'ordonnancement temps réel. En effet, un système informatique temps réel est assujetti à l'évolution dynamique d'un procédé à contrôler [35], pour nous le véhicule autonome sous-marin.

3.2.2 Applications temps réel

On qualifie de temps réel tout système informatique dont le fonctionnement est assujetti à l'évolution dynamique de l'état du système qu'il contrôle. On distingue dès lors deux parties dans l'application : le système informatique temps réel et le procédé auquel ce système est connecté et dont il doit contrôler le comportement [35].

Deux types de contraintes temporelles concernent une application temps réel. Il s'agit des contraintes strictes et des contraintes relatives. Les contraintes strictes concernent un système temps réel pour lequel une erreur au niveau du temps (ou faute temporelle telles que : non respect d'une échéance, arrivée d'un message après les délais, irrégularité d'une période d'échantillonnage, dispersion temporelle trop grande dans un lot de mesure, etc.) peut induire des dommages importants d'ordre matériel ou humain. Ainsi cette faute temporelle devient intolérable.

Les contraintes relatives concernent, quant à elles, les systèmes temps réel pour lesquels les fautes temporelles sont relativement tolérées.

Ces contraintes temporelles sont une des particularités des systèmes informatiques temps réel comparés aux systèmes informatiques classiques. Les systèmes temps réel doivent tenir compte et gérer ces contraintes. En plus de fournir un résultat correct à l'issue du calcul, ces résultats doivent être fournis à temps. L'exactitude du résultat obtenu n'est pas suffisante à elle seule si la contrainte de temps n'est pas respectée. Un résultat correct mais hors délai est un résultat faux [35].

Un système informatique temps réel peut être défini comme :

- un générateur cyclique d'actions qui capte la dynamique du procédé par échantillonnage périodique et qui lui envoie des commandes au même rythme (on parle aussi de système "synchrone") ;
- un système réactif qui répond instantanément aux stimuli provenant du procédé selon la dynamique de celui-ci ;
- une organisation qui couvre ces deux aspects en ordonnançant l'exécution de tâches périodiques ou apériodiques ; on parle alors de système asynchrone.

L'ordonnancement des tâches, dans un système asynchrone, est chargé de faire respecter les échéances des requêtes du procédé ainsi que la résorption des conséquences des pannes. Il protège, donc, les tâches primordiales pour le procédé contre les fautes temporelles.

3.2.3 Caractéristiques des tâches

Modèle canonique des tâches temps réel

Le modèle canonique d'une tâche temps réel est un modèle qui regroupe les principaux paramètres temporels qui caractérisent ces tâches, et guide l'ordonnancement temps réel. Selon ce modèle une tâche est définie par :

- des paramètres chronologiques qui dénotent des délais,
- et des paramètres chronométriques qui indiquent des dates.

Ces paramètres sont :

- r, sa date de réveil, autrement dit, le moment de déclenchement de la requête d'exécution ;
- C, sa durée d'exécution maximale lorsqu'elle dispose du processeur pour elle seule ;

- R, son délai critique, le délai maximal acceptable pour son exécution ;
- P, sa période lorsqu'il s'agit d'une tâche périodique ;
- d = r+R, c'est la date au-delà de laquelle la tâche commet une faute temporelle ;

Lorsqu'une tâche est périodique, ses dates de réveil successives, celles de chaque requêtes, sont :

$$r_k = r_0 + kP \tag{3.1}$$

avec r_0 : date de premier réveil et r_k : date du k^e réveil.

Ses échéances successives sont :

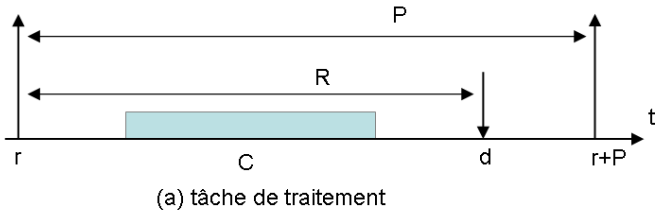
$$d_k = r_k + R \tag{3.2}$$

Si R=P, la tâche périodique est à échéance sur requête.

La qualité de l'ordonnancement sera d'autant meilleure que les valeurs de ces paramètres seront exactes ; aussi la détermination de ces paramètres est-elle souvent un aspect important de l'analyse et de la conception d'une application temps réel. En particulier, il faut savoir si on néglige ou non les temps de commutation de tâches, les durées d'utilisation des primitives du système d'exploitation, les durées de prise en compte des interruptions et les durées d'exécution de l'ordonnanceur. Si l'on ne peut les négliger, l'analyse conceptuelle doit les estimer et les inclure dans la durée d'exécution de la tâche.

D'autres paramètres, dérivés des paramètres de base, servent à suivre l'exécution de la tâche :

- $u = \frac{C}{P}$ est le facteur d'utilisation du processeur pendant une période,
- $ch = \frac{C}{R}$ est le facteur de charge du processeur,
- s est la date du début de l'exécution d'une tâche,
- e est la date de fin de l'exécution d'une tâche,
- $R(t) = d - t$ est le délai critique résiduel ou délai critique dynamique,
- $C(t)$ est la durée d'exécution résiduelle à la date t,
- $LN = R - C$ est la laxité nominale de la tâche et indique le retard maximal pour son début d'exécution s quand la tâche s'exécute seule,
- $LN(t) = R(t) - C(t)$ est la laxité nominale résiduelle ou laxité dynamique ; c'est le retard maximal pour reprendre l'exécution quant la tâche s'exécute seule ; on a encore $LN(t) = R + r - t - C(t)$
- $TR = e - r$ est le temps de réponse de la tâche,
- $CH(t) = \frac{C(t)}{R(t)}$ est la charge résiduelle



(a) tâche de traitement

Figure 3.1 – Modèle canonique d'une tâche

Tâches préemptibles

Ces tâches, une fois élues par le système temps réel, peuvent être interrompues pour libérer le processeur qui sera utilisé par une autre tâche.

Tâches non préemptibles

Contrairement aux tâches préemptibles, les tâches non préemptibles ne peuvent être interrompues après avoir été élues par le système temps réel. Parmi ces tâches nous pouvons citer les tâches s'exécutant sous le contrôle du mécanisme d'interruption et les tâches qui effectuent des entrées et sorties directement sur un bus sans passer par un organe intermédiaire appelé DMA. Ce sont des tâches, souvent appelées immédiates, dont l'exécution doit être atomique.

Dépendance des tâches

La notion de dépendance ou d'indépendance des tâches concerne les liens existant entre ces tâches, tels que le fait que ces tâches puissent interagir entre elles par la communication implicite de messages ou une relation explicite de synchronisation. Il peut en résulter une relation de précédence entre tâches. Cette relation est dite statique, car elle est connue a priori et n'évolue pas. Elle est représentée par un graphe de précédence.

Un autre point qui lie les différentes tâches concerne certaines ressources autres que le processeur. Par exemple les ressources dites exclusives ou critiques, imposent aux tâches de les utiliser en exclusion mutuelle.

3.3 Ensemble de tâches sur un processeur

A un moment donné l'ensemble des tâches en concurrence pour le processeur est appelé une configuration. Certains paramètres permettent de caractériser l'exécution d'une configuration sur un processeur :

soit C une configuration composée de n tâches $\{T_1, T_2, ..., T_n\}$

Facteur d'utilisation du processeur :

$$U = \sum_{i=1}^{n}(u_i) = \sum_{i=1}^{n}(\frac{C_i}{P_i}) \tag{3.3}$$

Facteur de charge du processeur :

$$CH = \sum_{i=1}^{n}(ch_i) = \sum_{i=1}^{n}(\frac{C_i}{R_i}) \tag{3.4}$$

3.4 Caractéristiques des algorithmes d'ordonnancement

3.4.1 En ligne ou hors ligne

Un algorithme d'ordonnancement hors ligne établit un échéancier des tâches dont il connaît tous les paramètres temporels. Ce type d'ordonnancement suppose de connaître

avant le début de l'exécution l'ensemble des tâches qui vont demander les processeurs. Cette approche n'est pas adaptée aux cas où des changements dus à des retards de tâches apparaissent durant l'exécution.

Un algorithme d'ordonnancement en ligne calcule à chaque instant la prochaine tâche à exécuter sur le processeur. Ce calcul se base sur les paramètres temporels des tâches prêtes à être exécutées. Ce choix peut être remis en cause lors de l'occurrence d'un événement nouveau sans que la date de cette occurrence n'ait à être connue à l'avance. Cette approche dynamique donne des solutions qui sont moins bonnes que celles de l'approche statique puisqu'elle utilise moins d'information et les surcoûts de mise en oeuvre sont plus importants. Mais elle permet l'arrivée imprévisible de tâches et elle autorise la création progressive de la séquence d'ordonnancement.

3.4.2 Préemptif ou non préemptif

Dans le cas d'un algorithme préemptif, une tâche élue peut perdre le processeur au profit d'une autre tâche jugée plus urgente ou plus prioritaire ; elle passe à l'état prêt pour attendre d'être (éventuellement) élue ultérieurement. Un algorithme préemptif n'est utilisable que si toutes les tâches sont préemptives.

Les algorithmes d'ordonnancement non préemptifs n'arrêtent pas l'exécution des tâches élues. Il peut en résulter des temps de réponse plus longs qu'avec un ordonnancement préemptif.

3.4.3 Période d'étude

L'étude pour valider une configuration de tâches périodiques conduit à faire une analyse temporelle de l'exécution de cette configuration. Quand les tâches sont périodiques, cette exécution dure indéfiniment. En fait, le comportement de la configuration est périodique et il suffit d'en analyser une période dite période d'étude, ou pseudo-période.

La période d'étude commence à la première date de réveil d'un tâche de la configuration :

$$\min_{1 \leq i \leq n} (r_{0i}) \tag{3.5}$$

Elle se termine à une date qui est fonction du plus petit commun multiple des Périodes P_i des tâches de la configuration :

$$\max_{1 \leq i \leq n} (r_{0i}) + PPCM(P_i) \tag{3.6}$$

3.5 Principales politiques

La politique d'ordonnancement du processeur détermine quelle tâche présente dans la file des tâches prêtes est élue. Nous allons décrire ci-dessous les principales politiques mises en oeuvre dans les systèmes classiques.

3.5.1 Premier arrivé, premier servi (FIFO)

Dans cette politique, l'unité de calcul est allouée aux tâches dans leur ordre d'arrivée. Les tâches de faibles durées d'exécution sont pénalisées lorsqu'elles suivent dans la file

d'attente une tâche longue.

3.5.2 Plus court d'abord (Shortest Job First)

Cette politique tente de remédier à l'inconvénient mentionné pour la politique précédente. Maintenant, l'unité centrale est allouée à la tâche de plus petit temps d'exécution.

3.5.3 Tourniquet (Round Robin)

On classe de façon arbitraire les tâches en attente du processeur dans une file. On alloue le processeur à la première tâche pendant une durée qu'on appellera **quantum**. Si la tâche se termine avant la fin du quantum de temps, elle libère le processeur et la tâche suivante dans la file d'attente est lancée selon le même principe. Si le quantum de temps vient à se terminer avant que la tâche ne finisse son exécution, cette dernière est préemptée et insérée à la fin de la file d'attente et le processeur est alloué de nouveau pendant un quantum à la tâche en tête de file.

Les performances de cette politique sont fonction de la taille du quantum de temps (généralement entre 10 et 100 ms). Un quantum trop grand a pour conséquence un grand temps de réponse et un quantum trop petit augmente le nombre de commutations de contexte.

3.5.4 Priorités constantes

Chaque tâche se voit attribuée une priorité qui n'évoluera pas durant la vie de la tâche. A un instant donné, c'est la tâche de plus haute priorité qui est élue. Le problème posé par cette politique est le risque de "famine" encouru par les tâches de faible priorité. Une des solutions est de faire "vieillir" cette priorité en l'augmentant en fonction du temps d'attente.

Rate Monotonic

L'algorithme Rate Monotonic attribue une priorité constante fonction de la période de la tâche. La tâche de plus petite période sera la tâche la plus prioritaire.

Inverse Deadline

Cet algorithme attribue la plus grande priorité à la tâche de plus petit délai critique.

3.5.5 Priorités variables

Les algorithmes à priorité variable calculent au cours du temps la priorité d'une tâche en fonction de ses paramètres temporels.

Earliest Deadline

Earliest Deadline accorde la plus grande priorité à la tâche dont l'échéance est la plus proche. Pour des tâches à échéances sur requête, une condition nécessaire et suffisante

d'ordonnancement est :

$$\sum_{i=1}^{n} \frac{C_i}{P_i} \leq 1 \tag{3.7}$$

Pour des tâches quelconques, une condition suffisante d'ordonnancement est :

$$\sum_{i=1}^{n} \frac{C_i}{R_i} \leq 1 \tag{3.8}$$

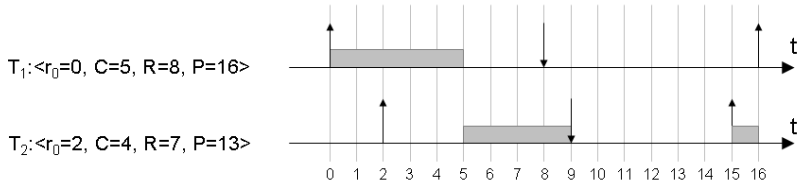


Figure 3.2 – Ordonnancement Earliest Deadline

La figure 3.2 illustre un exemple d'ordonnancement de deux tâches $T_1 = (r_0 = 0, C = 5, R = 8, P = 16)$ et $T_2 = (r_0 = 2, C = 4, R = 7, P = 13)$ par l'algorithme Earliest Deadline. A t=0, T_1 est la seule tâche réveillée, donc elle s'exécute seule. A $t = 2$, T_2 est réveillée mais si on compare les échéances des deux tâches ($d_{T_1} = 8$ et $d_{T_2} = 9$), on trouve que T_1 a l'échéance la plus courte alors T_1 aura la plus grande priorité. Par conséquent, T_1 continuera son exécution jusqu'à $t = 5$ où elle se termine. Ne reste alors que la tâche T_2 qui commencera son exécution à $t = 5$ pour se terminer à $t = 9$.

Least Laxity

L'algorithme Least Laxity donne la plus grande priorité à la tâche de plus petite laxité résiduelle $LN(t)$. La laxité résiduelle est le temps restant avant la date d'exécution au plus tard de la tâche.

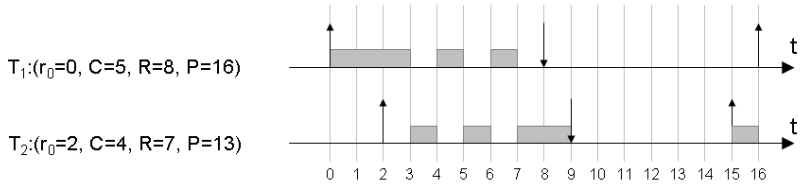


Figure 3.3 – Ordonnancement Least Laxity

La figure 3.3 reprend la même configuration de tâches que la figure 3.2 mais avec un ordonnancement Least Laxity. A l'instant 0, seule T_1 est réveillée : elle s'exécute jusqu'à

Table 3.1 – Laxité résiduelle des tâches T_1 et T_2

	$t=3$	$t=4$	$t=5$	$t=6$
LN_{T_1}	3	2	2	1
LN_{T_2}	2	2	1	1
Tâche élue	T_2	T_1	T_2	T_1

$t = 2$. A $t = 2$, T_2 est réveillée, on a alors deux tâches en concurrence pour le processeur. On calcule alors la valeur de la laxité résiduelle pour chacune des tâches :

$$LN_{T_1}(t=2) = R_{T_1}(t) - C_{T_1}(t) = (d_{T_1}) - C_{T_1}(t) = (8-2) - 3 = 3 \tag{3.9}$$

$$LN_{T_2}(t=2) = R_{T_2}(t) - C_{T_2}(t) = (d_{T_2}) - C_{T_2}(t) = (9-2) - 4 = 3 \tag{3.10}$$

On obtient la même laxité résiduelle on choisit de garder T_1 en exécution. Pour $t > 2$ on obtient les résultats suivants (tableau 3.1) : si les valeurs de laxité sont égales, on choisira arbitrairement la tâche T_1.

Les deux algorithmes Earliest Deadline et Least Laxity sont tous deux optimaux, mais le plus souvent Least Laxity occasionne plus de commutations de contexte que Earliest Deadline (figure 3.2 et 3.3).

Résumé du chapitre 3 :

- L'exécution d'une tâche est représentée par un modèle canonique caractérisé par quatre principaux paramètres : r sa date de réveil, C sa durée d'exécution, R son délai critique et P sa période.
- Nous citons cinq types de politiques d'ordonnancement : Premier arrivé - premier servi, plus court d'abord, tourniquet, priorités constantes et priorités variables.
- Nous trouvons deux principaux algorithmes à priorités variables : Earliest Deadline (accorde la plus grande priorité à la tâche dont l'échéance est la plus proche) et Least Laxity (accorde la plus grande priorité à la tâche de plus petite laxité résiduelle).
- Earliest Deadline et Least Laxity sont deux algorithmes optimaux. Earliest Deadline occasionne moins de commutations de contexte.

Chapitre 4

Positionnement

4.1 Contexte de l'étude

La robotique en général est un champ d'intégration qui regroupe des domaines de recherche différents et doit concilier les différentes solutions dans un tout cohérent. Le contrôleur du robot est à l'image de ce champ, un espace d'intégration de préoccupations différentes allant de considérations temps-réel à celles relatives à l'implantation d'algorithmes divers, à la gestion de contraintes techniques ou fonctionnelles ou encore à l'expression de missions. Les acteurs impliqués dans son développement ou son exploitation ont des intérêts scientifiques, et de fait des attentes et besoins, divers, qui en font d'ailleurs de potentiels "développeurs" ayant des expertises différentes...posant alors le problème d'accessibilité du contrôleur.

Le travail présenté s'inscrit clairement dans ce cadre d'étude et de mise au point d'une proposition de contrôleur, autrement appelée architecture de contrôle. Elle est certes une entité (une des parties constituantes d'un robot) gérant le comportement du robot mais elle est aussi, et avant tout peut-être dans notre contexte, un cadre d'intégration de travaux divers conduits au sein d'une équipe pluridisciplinaire, devant faciliter la concrétisation et le déploiement de solutions robotiques en réponse à des besoins scientifiques, industriels ou sociétaux comme en témoigne la variété d'applications de la robotique sous-marine exposée en première partie.

En tant qu'architecture de contrôle, la proposition doit évidemment permettre de répondre aux problématiques posées dans le cadre du contrôle des véhicules sous-marins (cf. prochaine section) mais elle doit également être acceptée et accessible par les différents utilisateurs potentiels. L'acceptation passe notamment par une proposition simple à appréhender, à manipuler, à faire évoluer...chacun doit pouvoir s'y "retrouver". Disposer pour cela d'un cadre, des abstractions et des outils nécessaires est une des attentes des acteurs (de l'équipe). Cette précision est importante car elle caractérise la nature de la proposition faite dans ce travail ; une structuration selon un modèle simple et clair, fondée sur des entités de base manipulables (indépendamment), composables facilement et contrôlables, et reposant sur un processus de développement qui ne soit pas complexe. L'accent est mis sur ces critères et pour autant nous n'avons eu que peu recours aux

dernières avancées du génie logiciel telles que les langages à composants ou les approches dirigées par les modèles par exemple, qui sans aucun doute favorisent modularité, réutilisabilité, indépendance à la plateforme cible, génération automatique de code, etc. Au regard de la complexité encore inhérente à ces propositions, à leur difficulté d'acceptation entre autres de par l'insuffisance d'outils pour les supporter ou les exploiter simplement, notre proposition s'est réduite à une formalisation moins poussée mais suffisamment évoluée pour répondre aux enjeux identifiés et surtout réalisable, et réalisée, dans le cadre de cette thèse. De toute évidence ce travail n'est qu'une étape dont la proposition ne doit pas inhiber l'évolution, mais elle doit être concrétisée, i.e. implémentée et utilisable.

4.2 Enjeux et orientations de notre étude

Les enjeux plus particulièrement relatifs à l'architecture de contrôle, autres que ceux mentionnés précédemment, comprennent :

- Le positionnement du robot sous-marin à la fois comme objet de recherches et comme vecteur d'exploration du milieu.
  Dans le premier cas, l'architecture de contrôle doit être ouverte pour permettre à chaque acteur d'intervenir sur "sa partie". L'automaticien par exemple doit pouvoir intégrer diverses lois de commande ou structures de commande intégrant la navigation et le guidage, voire diverses lois de coordination des véhicules au sein d'une flottille. Dans le second cas, le robot doit pouvoir être perçu comme un dispositif robotique évolutif à l'image des missions qu'il doit accomplir ; il s'apparente à une "sonde autonome" capable d'aller effectuer toutes sortes de relevés au fond de l'eau (cf. chapitre 1). Sa capacité à répondre au changement de charge utile (i.e. modification de l'instrumentation embarquée) repose en partie sur l'architecture qui doit permettre d'intégrer et rendre utilisable aisément un nouvel appareillage. . .donc de changer une "partie", aussi petite soit-elle (un seul capteur par exemple), dans l'architecture matérielle et logicielle.
  Dès lors que l'on évoque la notion de "partie", un besoin évident de modularité en découle. Modularité sous-entend module au sens général d'une entité logicielle bien définie, manipulable, composable. C'est une caractéristique quasiment commune aux architectures présentées (cf. chapitre 2). Il faut néanmoins adopter une définition simple, des règles de composition claires et offrir les mécanismes sous-jacents pour supporter les interactions résultantes de cette composition. Sur ce point, nous soulignons encore que les tendances en génie logiciel, dont les approches basées composants, s'attachent à une définition rigoureuse de ces interactions en explicitant les interfaces, les services et les ports des entités (composants), les connexions entre ces entités y compris les protocoles d'interaction sous-jacents, avec une formulation qui permet d'en vérifier la cohérence (conformité de composition). Nous nous limiterons à ce stade de nos travaux à définir clairement les entités de notre proposition d'architecture, leurs points de composition et à offrir un mécanisme supportant cette composition, i.e. supportant les interactions selon deux axes : flux de contrôle d'une part, et flux de données/événements d'autre part.
- La prise en compte d'une des particularités des véhicules sous-marins, les interférences potentielles entre ses instruments acoustiques.
  En effet comme nous l'avons évoqué au chapitre 1 les capteurs utilisés sous l'eau

posent souvent des problèmes d'interférences en cas d'utilisation simultanée. Certains de ces capteurs intervenant dans le processus de commande, la prise en compte explicite, et transparente (i.e. automatique), est nécessaire pour la sauvegarde de l'intégrité de l'engin commandé. Ces instruments doivent être considérés comme des ressources dont l'exploitation est contrainte. Ce problème n'apparaissant pas ou rarement en robotique terrestre, ces ressources et leur gestion ne sont pas réifiées dans les propositions d'architecture que nous avons présentées. Les ressources considérées sont plus souvent des ressources agrégées, i.e. de haut-niveau d'abstraction.

- La prise en compte des aspects classiques des systèmes embarqués temps-réel dont relève tout robot d'exploration.
  La gestion explicite des contraintes temporelles, le respect desquelles étant primordial notamment pour préserver la validité des équations de l'automatique assurant la stabilité de la commande de l'engin. Il est évident qu'il faut respecter les échéances temporelles précises sur lesquelles repose la bonne exécution de la structure de commande. Si, par exemple, la structure de commande (mesures capteurs-loi de commande-applications actionneurs) et la période de la boucle de contrôle doivent pouvoir être indiquées par l'automaticien, l'ordonnancement de l'exécution des tâches correspondantes doit être transparent au même titre que la gestion des capteurs que l'automaticien implique dans sa structure de commande, et ce pour éviter l'interférence potentielle entre ces instruments. Transparent ne signifie pas occulte ; au contraire l'ordonnanceur doit apparaître comme une entité à part entière de l'architecture, son rôle et son pouvoir décisionnel doivent être clairement définis, son algorithme modifiable sans impact sur les autres entités de l'architecture. La gestion des contraintes qu'il assure doit être explicite, dont l'exploitation des ressources relatives aux instruments embarqués précédemment mentionnée, etc. C'est une entité rarement réifiée dans les différentes propositions d'architecture que nous avons présentées. Lui donner existence impose de lui donner aussi les moyens de contrôler l'activité des entités dont il doit maîtriser l'exécution. . .cela influence donc la définition des "modules", ici au sens des interactions relatives à son activité. Nous avons très brièvement exposé la formulation classique sur laquelle repose l'ordonnancement (cf. chapitre 3) ; il s'agit d'une problématique grandement traitée dans la littérature, qui concerne d'ailleurs de nombreux domaines. Pour espérer intégrer dans notre architecture la stratégie d'ordonnancement la plus adéquate, et bénéficier des avancées dans ce domaine, il est essentiel qu'elle soit supportée par une entité dédiée dans l'architecture et qu'elle soit basée sur un modèle canonique communément admis.
- Une organisation du contrôleur, de ses entités constituantes, répondant aux besoins de structuration et de réactivité.
  Nous avons présenté un ensemble d'architectures développées pour la robotique en général et certaines pour la robotique sous-marine en particulier. La conciliation du besoin de structuration et de celui de réactivité se reflète à travers des approches dites mixtes ; c'est-à-dire des architectures alliant une hiérarchisation en niveaux tout en offrant des mécanismes favorisant la capacité de réaction aux "perturbations" à différents niveaux (cf. chapitre 2). L'organisation et ses critères doivent être clairs, la simplicité de prise en main par les acteurs en dépend notamment. Une organisation simple peut relever de deux niveaux, l'un dit exécutif pour gérer

l'exécution des modules (ou séquences de modules) en répondant aux problèmes du temps-réel, et l'autre qualifié de décisionnel supportant la sélection (l'engagement) du ou des comportements à exécuter dans le contexte courant, i.e. le choix des modules (ou séquences de modules) à exécuter. Cette organisation ne doit pas inhiber la possibilité de mise en place de modèles architecturaux différents, i.e. relevant d'approches qualifiées de hiérarchisées (NASREM[23] par exemple) et celles qualifiées de comportementales (Subsumption[20] par exemple). Un raffinement du niveau dit décisionnel en plusieurs "sous-niveaux" doit également être possible. Cette structuration en deux niveaux, sur laquelle s'appuie notre proposition, repose en fait sur deux mondes différents ; le monde "périodique" au sein duquel s'exécutent les modules et le monde "événementiel" dont relève la prise de décision. Le fonctionnement est différent dans ces deux mondes. De plus, à chaque monde son espace de représentation, plus ou moins agrégé, plus ou moins abstrait ; par exemple une gestion de ressources différentes, des ressources bas-niveau telles que les capteurs et des ressources haut-niveau tels que des sous-systèmes.

– Une proposition d'architecture qui soit interfaçable avec un simulateur.
Le développement des commandes et leurs réglages se fait souvent de façon expérimentale, au fil d'essais réalisés en conditions réelles...avec toute la "lourdeur" des aspects expérimentaux dans ce domaine de la robotique sous-marine, qui nécessite même pour de petits véhicules toute une organisation (aller en mer, mettre le bateau à l'eau...). Nous avons largement évoqué la nécessité de mettre à disposition des acteurs concernés une plateforme logicielle facilitant l'évolution (la modification) du contrôleur. Sur un autre plan, pour faciliter les tests et les réglages, ne serait-ce que pour évaluer qualitativement la validité de paramètres temporels de tâches (ensemble de modules) ou l'apport d'une nouvelle loi de commande, il serait intéressant de pouvoir avoir recours à la simulation tout en exécutant cette commande dans l'univers du contrôleur temps-réel. Si la structuration du contrôleur est claire, substituer un simulateur aux capteurs et actionneurs (et à l'environnement bien sûr) ne devrait pas être ardu. Le travail présenté n'aborde pas la simulation, ni les simulateurs ; cette remarque témoigne simplement de la prise en compte de cet aspect dès la conception, a fortiori dans ce contexte de la robotique sous-marine où l'étude de la coordination de flottilles par exemple nécessite sans aucun doute de pouvoir avoir recours à la simulation tout en restant dans l'univers réel et contraint de la commande temps-réel.

Répondre à tous ces problèmes (enjeux) en les considérant indépendamment les uns des autres est sûrement possible, mais la formulation et la mise en oeuvre d'une solution globalement cohérente nous semble imposer plutôt une approche adressant l'architecture dans sa globalité, en considérant concepts et implantation. Nous allons donc proposer une architecture de contrôle en se focalisant sur les aspects logiciels. Notre contribution doit prendre en compte les différents problèmes évoqués, doit proposer une solution fondée sur des concepts et/ou formulations clairs, la mettre en oeuvre et l'accompagner du minimum d'outils nécessaires à sa "prise en main".

Nos travaux étant positionnés, nous allons maintenant les détailler dans la seconde partie de ce manuscrit.

Deuxième partie

Proposition d'une architecture pour la gestion contextuelle de tâches dans le cadre du contrôle d'un véhicule sous-marin autonome

Introduction

Après avoir présenté le cadre de notre étude sous ses divers aspects, nous allons mener quelques réflexions sur la construction d'une architecture logicielle pour la robotique sous-marine. Le critère principal qui va guider la conception est la modularité, avec une représentation explicite des flux de données et de contrôle. Cette modularité consiste à concevoir le contrôleur (logiciel) sous la forme d'un ensemble d'entités. Parmi nos objectifs figure celui de l'accessibilité à tout ou partie, et ce à des niveaux d'abstraction, d'intérêt scientifique ou de compétences différents.

Nous présenterons dans un premier chapitre, la déclinaison que nous faisons de ces entités logicielles, que nous appellerons ici des "**modules**" (conception, implémentation etc.)

Le respect des règles de construction des modules, selon un principe similaire à l'instanciation d'un "modèle" contribue à conférer à notre architecture des propriétés d'évolutivité, de réutilisabilité de tout ou partie, etc.... Il ne s'agit pas d'ingénierie dirigée par les modèles - IDM (ou MDE - Modele Driven Engineering) [36], méthodologiquement plus poussée, mais de définir un "modèle" en tant que référence de description avec des points d'interaction simplement et clairement définis.

D'autre part les aspects relatifs au fonctionnement du logiciel tels que la réactivité et la robustesse, seront traduits par la manière d'agencer (hiérarchisation) les modules, la manière de gérer les différents flux de contrôle et de données ainsi que par la répartition non ambiguë des rôles de chacun. Ces points seront alors évoqués dans un deuxième chapitre et une proposition d'organisation sera faite. Même si l'agencement de ces entités, modules, repose ici sur une certaine hiérarchisation, d'autres agencements (donc architectures) seraient possibles.

La gestion de l'instrumentation de bord est une étape incontournable pour les logiciels de contrôle robotique, nous aborderons dans un dernier chapitre les interactions logiciel/matériel et leurs contraintes. Au regard de cette étude nous détaillerons dans un troisième chapitre une solution apportée pour gérer cette instrumentation.

Chapitre 5

Principes de base de conception d'une architecture logicielle de contrôle

5.1 Introduction

Dans un robot, on distingue une partie opérative et une partie contrôle. La partie opérative ou partie physique, est composée de capteurs et d'actionneurs ainsi que d'éléments mécaniques. La partie opérative doit bien sûr être adaptée aux tâches (missions) à effectuer (une pince pour la préhension, un propulseur pour le déplacement dans un liquide, etc.). Les capteurs, qui peuvent être extéroceptifs ou proprioceptifs, vont permettre de rendre compte de l'état de l'environnement, de l'état du robot et de l'état du robot par rapport à son environnement. La partie contrôle, souvent appelée contrôleur, est le centre de décision du robot. Il intègre les ordres de l'opérateur (la mission) ainsi que l'état du système et celui de son environnement pour prendre les décisions (traduites par la commande des actionneurs) qui vont diriger le robot vers l'accomplissement de sa mission tout en restant en adéquation avec le contexte. Le contexte est donc caractérisé par le but à atteindre, l'état du robot et les contraintes imposées, perçues de manière proprioceptive ou extéroceptive (niveau d'énergie, obstacles par exemple).

Le contrôleur du robot est supporté par un ensemble de matériels électroniques (mémoires, processeurs, liens de communications, etc.). Il est de plus en plus fréquent de trouver des composants électroniques reconfigurables (dont la structure interne est modifiable par programmation) mais la partie centrale d'un contrôleur (siège des décisions "haut niveau") est dans la plupart des cas constituée d'une ou plusieurs cartes processeurs (ou microcontrôleur ou encore coeur de processeur sur composant électronique programmable) pouvant exécuter des algorithmes stockés en mémoire. La majeure partie des fonctionnalités du contrôleur est alors le fruit de l'exécution de ces algorithmes. L'ensemble de ces algorithmes, appelé logiciel de contrôle/commande, implémente les mécanismes permettant d'utiliser le matériel embarqué (capteurs, actionneurs, matériel informatique) pour accomplir la mission confiée.

Le contrôleur d'un robot doit être en mesure de commander la partie opérative pour accomplir les objectifs de la mission en prenant en compte les contraintes matérielles (capteurs interférents, butées des actionneurs) et en veillant à être en adéquation avec l'état

du système (dont la charge de l'unité de calcul par exemple) et l'état de l'environnement (obstacle). La prise de décisions, dont résulte le comportement du robot (déplacement, préhension), doit être en adéquation avec la dynamique du système contrôlé, c'est-à-dire effectuée dans un intervalle de temps tel que le système physique n'ait pas eu le temps de significativement évoluer. Pour cela, le contrôleur doit être temps réel ; il doit par exemple établir de manière périodique (selon une période dépendant de la dynamique du système physique) les commandes des actionneurs, mais également être réactif à l'occurrence d'événements.

Dans de tels systèmes, le contrôleur s'avère être un organe de plus en plus complexe. La coexistence de phénomènes périodiques et sporadiques est caractéristique. En effet, ce dernier doit exploiter périodiquement le matériel de bord, prendre les décisions qui vont orienter la mission dans un environnement non connu a priori et gérer ses propres ressources (énergie et temps processeur par exemple).

Dans un projet robotique, la conception d'un contrôleur est une phase importante, tant du point de vue matériel que du point du vue logiciel. En effet, l'ensemble des composants matériels du contrôleur doit être capable d'offrir l'ensemble des connexions permettant d'exploiter l'instrumentation (récupération des données capteurs, envoi des commandes aux actionneurs) ainsi que les filtres et les circuits logiques de traitement de l'information. Le logiciel qui exprime généralement le coeur de la prise de décision du robot doit être capable d'exploiter l'ensemble du matériel pour mener à bien la mission.

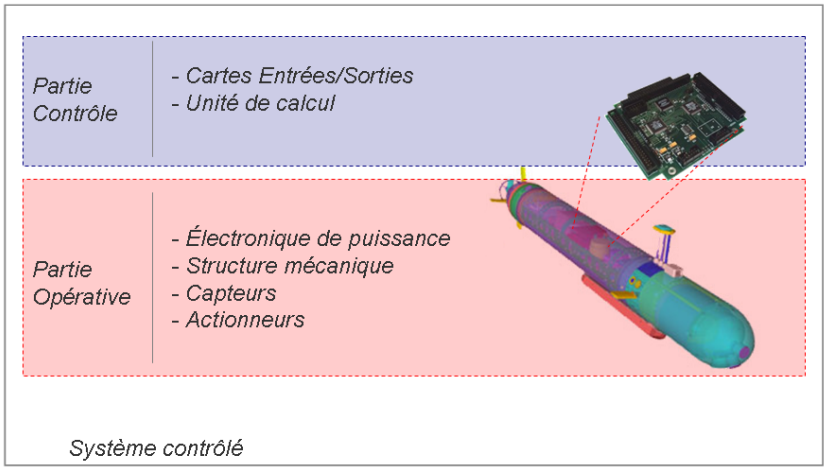


Figure 5.1 – Robot sous-marin autonome Taipan II

Agrégation de composants matériels parfois hétérogènes et de traitements divers, le contrôleur voit sa conception commencer par l'établissement de son architecture, c'est-à-dire les entités constitutives du contrôleur et leur interconnexion tant sur un plan logiciel que matériel. Sur un plan logiciel, ces entités supportent différentes fonctions plus ou

moins élémentaires (fournies par un matériel ou par l'exécution d'un algorithme). L'assemblage de ces entités, leur interconnexion à la fois en terme de flux de données et de flux de contrôle, permet de créer les fonctionnalités "avancées" ou "évoluées" nécessaires (ou requises). La complexité d'un contrôleur se répercutant sur son architecture, le développement de celle-ci impose de définir cette entité logicielle, et les règles de fonctionnement et d'interconnexion facilitant la mise en oeuvre, la maintenance mais également favorisant l'évolutivité et la réutilisabilité des éléments. L'ensemble de ces règles peut se décliner sous la forme d'un patron, appelé modèle d'architecture, à suivre lors de l'implémentation et des différentes modifications de l'architecture du contrôleur [19]. La définition de ces règles, donc la construction du modèle d'architecture, est dirigée par certains critères émanant du besoin de performance dans le fonctionnement du contrôleur (respect des contraintes de temps pour la commande, robustesse face aux données capteurs erronées etc.) ainsi que des besoins de simplicité dans ses différentes phases de spécification, d'implémentation, de maintenance et d'évolution.

Au niveau matériel, la structure du contrôleur des véhicules de petite taille est très souvent centralisée autour d'un organe central programmable. Le programme ou logiciel s'exécutant sur ce composant, est le siège de toutes les décisions prises au sein du contrôleur. Le reste du matériel étant consacré soit à des traitements lourds de capteurs évolués (flux vidéo d'une caméra par exemple), soit à des traitements plus "basiques" tels que le filtrage et la mise en forme de données provenant, ou à destination, de l'instrumentation de bord. La majeure partie de la conception d'un tel contrôleur, hors considération matérielle, repose sur la conception du logiciel central. Ce dernier est en relation avec un certain nombre d'éléments matériels différents et est alors composé d'un ensemble de traitements qui devraient être indépendants. On étudiera alors l'architecture de ce logiciel qui mettra en évidence les différents éléments du logiciel, leurs relations et leurs liens avec le matériel. L'architecture est conçue selon un ensemble de propriétés fondamentales à savoir la modularité, l'évolutivité et la réutilisabilité.

Dans un premier temps, nous exposerons la manière dont nous exprimons les flux de données qui permettent d'articuler les entités de l'architecture entre-elles et dans un second temps nous introduirons les flux de contrôle.

5.2 Modularité

Un logiciel de contrôle, intègre plusieurs fonctionnalités (filtrages de données, prises de décisions, calcul de lois de commande, planification etc.) de natures différentes. Celles-ci doivent être assez indépendantes dans leur exécution ; les liens d'échange de données, de contrôle ou de synchronisation entre-elles doivent être limités et clairement identifiés. Le manque d'indépendance des éléments constitutifs d'un contrôleur ne fait qu'accroître sa complexité. Ainsi pour faciliter le développement d'un contrôleur, on décompose ce dernier en entités (qu'on appellera ici des modules) agrégeant un ensemble de traitements et de données de manière cohérente selon une encapsulation permettant notamment de minimiser les interdépendances et de les mettre en évidence.

Chaque décomposition doit veiller à séparer les éléments présentant le moins d'interactions pendant leur fonctionnement et va rassembler les éléments fortement liés. Nous serons amenés à un compromis en terme de granularité notamment pour des raisons d'évolutivité et de réutilisabilité. Cette décomposition du logiciel de contrôle en modules

permet de mettre au point et également de tester ces briques, indépendamment les unes des autres.

Réifier les dépendances entre les entités, à la fois en terme de flux de contrôle que de flux de données, permet de "formaliser" la conception du contrôleur comme un assemblage d'entités. Cette démarche de sub-division d'un problème (ici un logiciel) en plusieurs sous problèmes facilite le développement. Chaque sous problème est résolu par un module qui a un nom, des données initiales et éventuellement des données résultats. Ces modules peuvent être "appelés" pour exécuter la tâche (au sens de fonctionnalités) pour laquelle ils ont été écrits. On ne se soucie pas du "comment" est réalisée la tâche, au sens de l'algorithme implanté dans le module, mais uniquement de la tâche qu'il réalise avec ses flux d'entrées et/ou de sortie. L'ensemble des modules résolvant les sous problèmes ainsi que leur assemblage forment le logiciel correspondant au problème initial. D'autre part, notre décomposition n'est pas uniquement orientée selon les fonctionnalités (au sens de traitements informatiques) mais aussi selon les ressources, qu'elles soient physiques ou abstraites. Ainsi dans notre approche les instruments de bord du véhicules sont, dans la mesure du possible, gérés par des modules différents.

Outre la facilité de développement apportée par cette approche classique (division du problème en plusieurs sous problèmes), elle doit permettre aussi à plusieurs développeurs de travailler indépendamment les uns des autres puis d'assembler leurs réalisations en un seul tout final, sous réserve d'avoir retenu un même "format" de données (standardisation) [5].

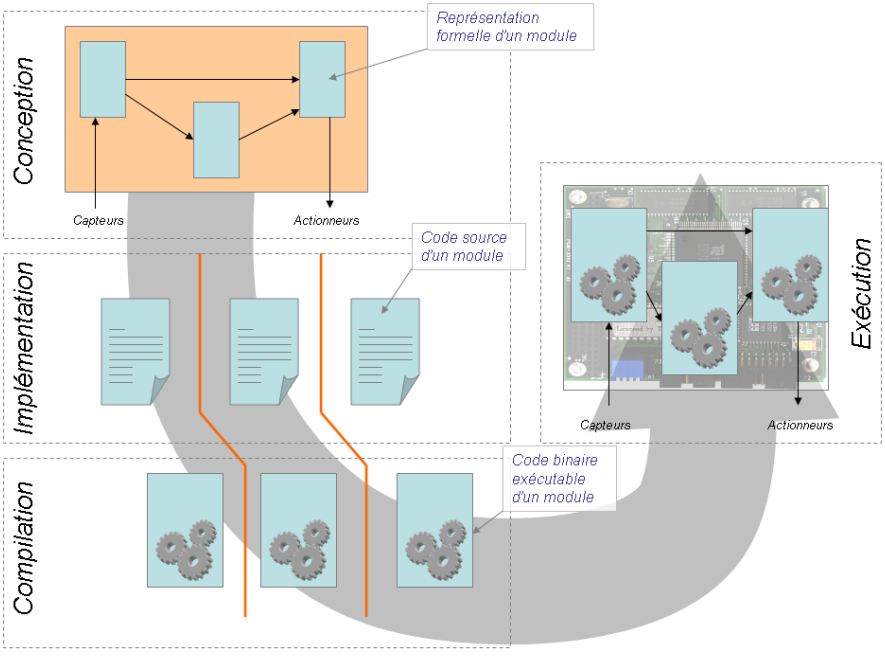


Figure 5.2 – Processus de développement des modules

Dans une architecture logicielle, la réalisation complète d'une tâche donnée (au sens robotique du terme), comme par exemple le suivi de trajectoire, requiert plusieurs modules. Les différentes phases de la tâche sont exprimées par l'exécution des ces modules

qui implémentent chacun les algorithmes propres au rôle qu'ils jouent dans la réalisation de la tâche (acquisition d'une mesure capteur, calcul de la loi de commande, application de la commande aux actionneurs).

Les interfaces de chaque module (ensemble des interactions proposées par un module) comprennent deux parties : l'une relative aux interactions de contrôle d'activité d'un module (flux de contrôle) et l'autre relative aux interactions en terme d'échange de données et d'événements (flux de données). Le respect de ces interfaces lors de l'implémentation permet d'aller, pour un même module, jusqu'à une phase très avancée du développement (production d'exécutable par exemple) sans nécessiter de connaissance sur les algorithmes implémentés par les autres modules.

Cela permet de développer en parallèle plusieurs modules et les assembler dans une dernière étape, sur une même "infrastructure" (i.e. environnement "système" fournissant les services d'exécution). Outre le gain de temps dû à la parallélisation des développements informatiques, cette approche permet aussi à plusieurs ingénieurs d'intervenir dans le développement d'un logiciel de commande. Cela est d'autant plus important que le développement de contrôleurs en robotique relève d'acteurs aux compétences et rôles différents.

Un "*middleware*" (infrastructure d'exécution) commun à toutes les implémentations doit également être offert aux différents acteurs. Ce "*middleware*" constitue une abstraction du système d'exploitation facilitant la portabilité d'un logiciel de contrôle d'une plateforme à une autre (pas d'appel système explicite dans les modules). Mais ce "*middleware*" offre avant tout une API qui met à disposition des programmeurs un ensemble de mécanismes systèmes utilisables tant dans l'implémentation des modules que dans leur composition (communication avec les périphériques, communication inter module etc.).

5.2.1 Au niveau de l'exécution

La figure 5.3 illustre la décomposition d'un exemple de schéma de commande en modules. Un module capteur (noté 1) se charge d'interroger l'instrumentation de bord pour en déduire la valeur de la position $P(x_m, y_m, z_m)$ du système. Un module loi de commande (noté 2) implémente la loi de commande du système ainsi qu'un comparateur. Ce module reçoit en consigne la position désirée $P(x, y, z)$ ainsi que la position mesurée fournie par le module 1, afin de générer la commande U(v,θ_1,ϕ_1). Cette commande sera envoyée à un autre module qui se charge de l'appliquer aux actionneurs du véhicule.

On comprend dans ce schéma que l'ordre d'exécution des modules joue un rôle important. Si le module 2 venait à s'exécuter avant le module 1, la valeur qu'il produirait serait une commande basée sur une observation capteur erronée ou retardée dans le cas où une exécution a déjà eu lieu. Une première solution est de synchroniser les modules entre eux, c'est à dire que l'exécution du module 2, par exemple, ne puisse se faire que si celle du module 1 s'est achevée. Si on intègre cette contrainte au niveau des modules eux-mêmes, on introduit un lien de synchronisation entre les modules qui brise la modularité. Cela est illustré par le cas où l'on serait amené à remplacer le module capteur 1 par un autre module (changement de capteur ou utilisation de plusieurs données capteurs pour le calcul de la position par exemple). Pour que l'ordre d'exécution soit respecté, on doit alors intervenir au niveau du module 2 pour lui indiquer de quel module il doit attendre les informations. La modification d'un module dans ce cas implique la modification d'un

autre. D'autre part si le module 1 se trouve arrêté par une erreur quelconque, toute la chaîne de modules dont il est le premier maillon se retrouve bloquée.

Nous dissocions les interactions entre les modules en deux parties. D'une part les flux de données et d'autre part les flux de contrôle. Les flux de données représentent l'ensemble des échanges de données ou d'événements entre les modules. Si on considère un même ensemble de modules, les flux de données qui vont les articuler peuvent être amenés à changer d'une mission à l'autre ou d'une architecture à l'autre. Les flux de contrôle commandent l'activité des modules. Dans notre approche, flux de données et flux de contrôle doivent reposer sur un ensemble de mécanismes d'un *middleware*, permettant de supporter des flux établis de façon "dynamique". Ainsi les flux de données et de contrôle peuvent être établis selon une configuration dépendante du contexte d'exécution (tâche robotique en cours par exemple).

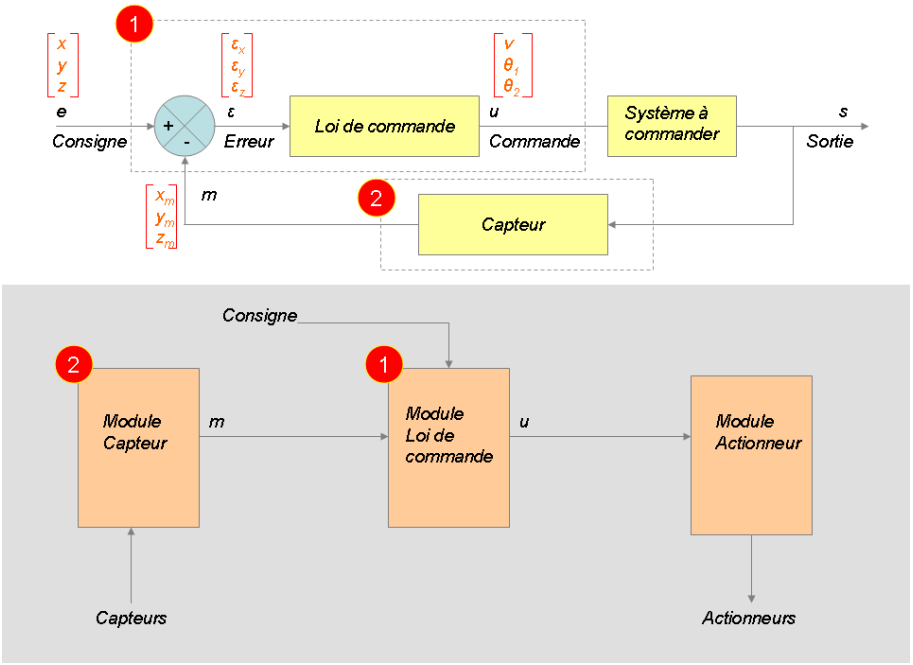


Figure 5.3 – Décomposition d'un schéma de commande sous forme de modules (flux de données)

5.2.2 Au niveau des interactions

Les modules obtenus à l'issue de la décomposition ne sont pas strictement indépendants. Un module renferme un ou plusieurs algorithmes manipulant un ensemble de variables dont certaines doivent être partagées avec d'autres algorithmes (appartenant à d'autres modules). Ainsi pour assurer le bon fonctionnement de l'assemblage final (c'est-à-dire du logiciel du contrôleur), il est indispensable de mettre en place différents liens logiques (supportant les flux de données et de contrôle) mettant en relation les modules entre eux et avec le matériel.

Une attention particulière doit être prêtée dès la conception, à ces liens logiques au

risque sinon de compromettre la modularité. Ces liens peuvent être implémentés en suivant différentes approches. La plus courante est le partage de variables. Une variable désignée par un nom unique peut être consultée et modifiée par un ensemble de modules. Le module qui va la modifier sera appelé écrivain et celui qui va la consulter sera le lecteur. Ce type de mécanisme (variables partagées) impose à deux modules de toujours conserver le même identifiant (nom de variable ou adresse par exemple) pour la variable concernée. En cas de changement de l'identifiant, il faut répercuter la modification sur tous les modules manipulant cette variable. Le développement d'un module affecte alors le développement des autres.

Une autre approche est la transmission de données. Elle consiste à envoyer, c'est à dire recopier à un autre endroit, les données que l'on souhaite transmettre.

Dans notre approche, nous retenons le concept de transmission de données. Dans ce concept, les données sont dupliquées entre l'expéditeur et les destinataires. Il est à noter ici que la duplication des données permet le découplage temporel des algorithmes qui seront amenés à les manipuler.

Pour désigner les variables offertes et requises par un module, on leur associe un port. L'établissement d'une communication pour l'échange de données entre deux modules revient alors à relier un port de sortie de l'un à un port d'entrée de l'autre. Ces mécanismes sont supportés par un "*middleware*" commun sur lequel repose l'implémentation des modules.

Parmi les données mises à disposition par un module, certains modules peuvent en requérir la totalité ou bien juste une partie. Ainsi si toutes les variables sont fournies dans un même port, le flux de données sera surdimensionné pour certains modules qui n'en requièrent qu'une partie. Par exemple si un module A calcule la position et l'orientation du système piloté, on peut le concevoir de façon à ce qu'il produise sur un premier port les trois variables de la position et sur un second port les trois variables de l'orientation. Cela permettra à un module B de récupérer les variables de position de la part du module A et les variables d'orientation d'un autre module.

Le découpage de l'ensemble des données produites par un module sera orienté par l'utilisation qui va en être faite dans l'architecture logicielle. On peut s'attendre à ce que la position et l'orientation soient utilisées séparément.

On pourrait aussi décomposer totalement les variables produites par un module, de la position on obtiendrait les trois variables x,y et z et de l'orientation les trois variables ϕ, θ et ψ. Cette décomposition a pour avantage de rendre indépendantes toutes les variables (en terme d'entités échangeables) mais aura un impact sur les performances (nombre de communications) et augmentera en complexité la composition des modules. Elle sera justifiée si l'application met en oeuvre plusieurs modules qui vont requérir ces variables indépendamment les unes des autres. La décomposition des variables en groupes devra être effectuée au regard de l'utilisation qui en sera faite au sein de l'architecture. Une décomposition fine favorise la modularité mais alourdit la communication et une décomposition agrégée allège la communication mais limite la modularité.

Le système robotique sera souvent amené à effectuer plusieurs tâches (au sens robotique) donc à mettre en oeuvre différentes fonctionnalités. Ces fonctionnalités vont requérir des modules différents. On aura alors une configuration dynamique des modules (i.e. jeu de modules actifs et leur séquence) qui va de fait impliquer des flux de données dynamiques. En effet, un module A peut être en communication avec un module B dans

une certaine fonctionnalité, et avec un module C pour une autre fonctionnalité. On doit alors adopter un modèle supportant cette possibilité d'établir les flux de données de façon dynamique (voir section 5.3).

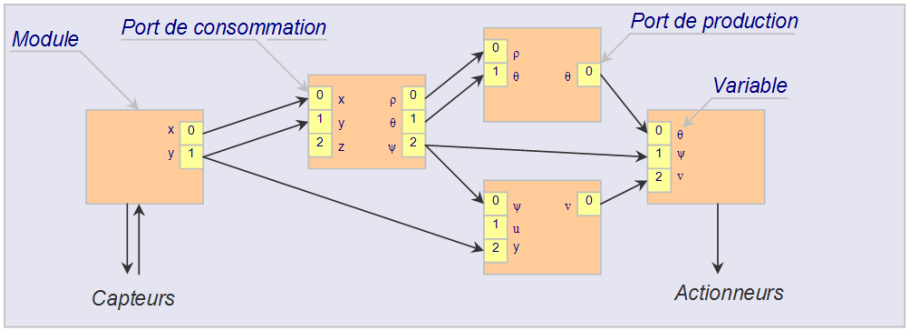


Figure 5.4 – Décomposition en modules

5.3 Evolutivité

Un contrôleur est amené à évoluer au cours du temps à l'image de l'évolution technologique et algorithmique. Il doit alors pouvoir être adapté facilement, on parle alors d'évolutivité du contrôleur. L'évolutivité doit pouvoir se faire à trois niveaux :

- 1. pouvoir ajouter et retirer des modules,
- 2. pouvoir modifier les modules,
- 3. pouvoir ré-agencer les modules en terme de flux de données et de contrôle.

Les points 1 et 2 sont assurés par les mécanismes cités à la section 5.2, requis d'autre part pour le critère de modularité. Le réagencement des modules, consiste à relier différemment leurs ports. Pour assurer cela, les informations concernant la liaison des différents ports des modules de l'architecture ne doivent pas être portés par les modules eux-mêmes mais être définis à l'extérieur. L'assemblage des modules peut alors être décrit par une configuration qui peut évoluer selon le contexte (phases de la mission par exemple).

5.3.1 Au niveau de la conception

La conception d'un logiciel de commande doit supporter l'évolution induite par l'exploration de nouveaux domaines d'application ou par des évolutions au niveau du matériel ou du système d'exploitation. Ces évolutions vont introduire de nouvelles possibilités ou des restrictions que le logiciel devra suivre pour profiter des performances offertes, ou du moins rester compatible.

Supposons que l'utilisateur veuille tracer l'activité du logiciel, c'est à dire récupérer à un moment donné les dates d'exécution des différents modules constituant le logiciel. Pour ne pas avoir à modifier chaque module afin qu'il puisse offrir ce service, opération coûteuse en temps et réalisable uniquement lorsqu'on dispose des éléments logiciels sous une forme modifiable (code source par exemple), nous allons intervenir seulement sur un "*middleware*" commun. "*middleware*" sur lequel repose le développement de chaque

module et qui offre les principales fonctionnalités d'exécution et de communication d'un module.

En effet, dans notre approche nous distinguons d'une part le "modèle" du module et d'autre part l'"instance" du module. Le modèle va renfermer tous les mécanismes communs à tous les modules (envoi/réception de donnée, exécution/suspension etc.) et l'instanciation va consister à paramétrer ces mécanismes et ajouter le code propre utilisateur. En distinguant dans le squelette d'un module une partie "système" et une partie "utilisateur" (figure 5.5), nous pouvons construire une bibliothèque contenant toutes les fonctionnalités système (dont le point d'entrée principal). L'instanciation se fait alors à l'édition de liens avec les codes utilisateur. Lorsque les deux codes sont distincts, la modification d'une caractéristique générale, commune à tous les modules, nécessitera seulement l'intervention au niveau de la bibliothèque de fonctions (renfermant les services de la partie "système" des modules) puis la recompilation de tous les modules.

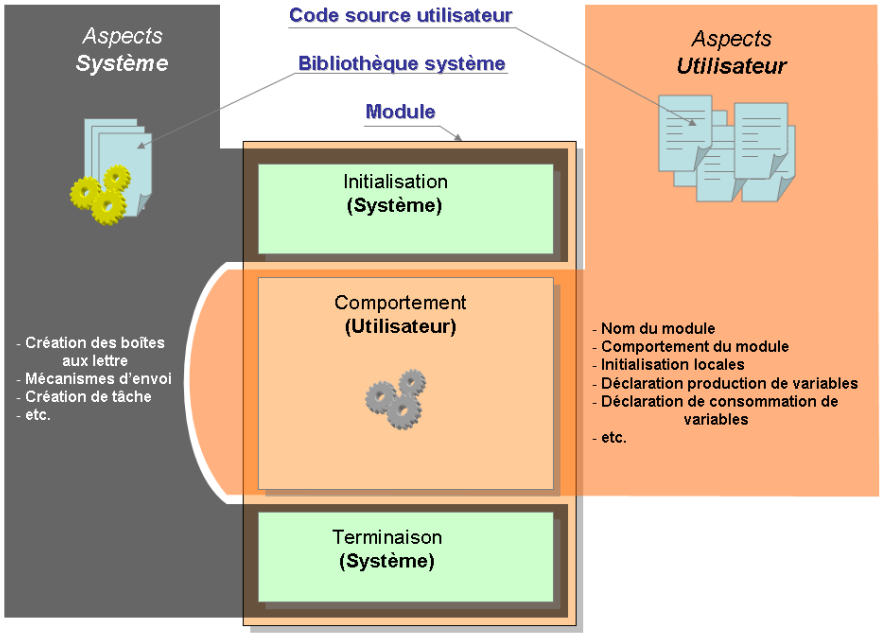


Figure 5.5 – Structure d'un module

5.3.2 Au niveau de l'implémentation

On rencontre trois principaux types de besoin d'évolution qui vont induire une modification au sein d'une architecture logicielle de commande :

- l'**évolution algorithmique** qui se traduit par une opération au niveau du code "utilisateur" d'un module,
- l'**évolution matérielle** qui traduit une modification de l'instrumentation du système commandé et qui va induire la modification ou le remplacement complet d'un module,
- l'**évolution structurelle** qui se traduit par une modification du schéma d'interconnexion des modules déjà présents.

L'exemple le plus courant au sujet de l'évolution algorithmique est celui de la loi de commande déjà implémentée mais qui doit faire l'objet de réglages empiriques (réglages des gains pour un PID, si ces derniers ne sont pas déclarés comme des paramètres modifiables de l'extérieur) ou encore un changement dans la répartition des poids à l'intérieur d'un véhicule sous-marin peut rendre caduc une loi de commande (la commande utilise généralement un modèle physique du système commandé qui intègre la répartition de sa masse sur son volume). Ce type d'intervention ne concerne que le module et plus précisément un algorithme dans un module (on ne considère pas ici le cas où la modification induit une modification dans les variables produites par le module). Ce type de modification est transparent et la structure de l'architecture le permet sans autre type de modification. On peut noter toutefois que même si la modification d'un algorithme n'affecte pas nécessairement l'interface d'un module, elle peut affecter son comportement. En effet, la modification de l'algorithme peut le rendre plus gourmand en temps de calcul. Ces problèmes seront traités plus en détails dans la section 7.3.

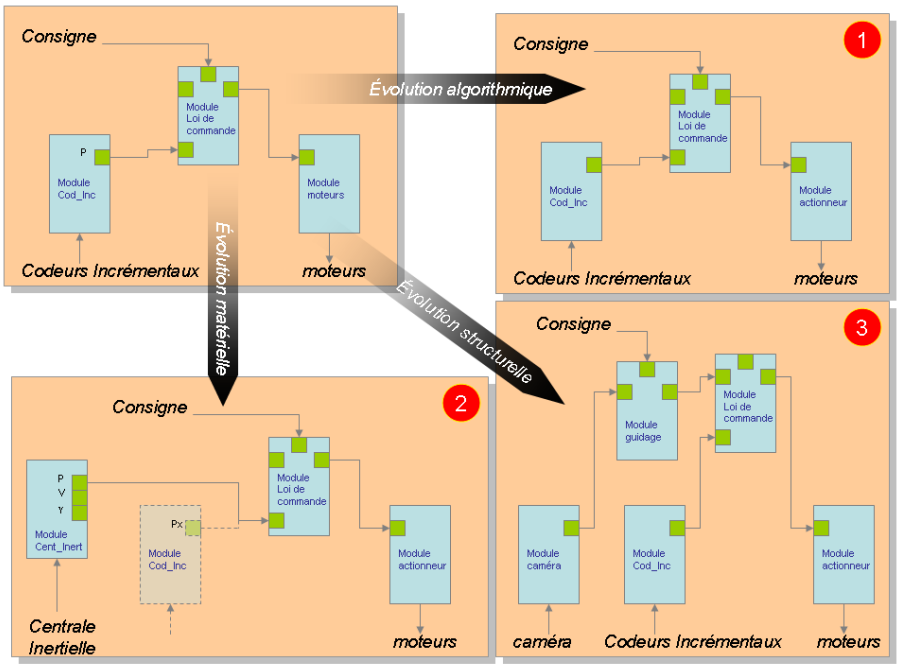


Figure 5.6 – Evolutivité de l'architecture

En ce qui concerne la modification d'un matériel de mesure ou d'actionnement, cela nécessite dans la grande majorité des cas la modification totale du code d'exploitation (ensemble des algorithmes et/ou pilotes mis en jeu dans le dialogue avec la partie matérielle concernée) et peut s'accompagner aussi d'une modification de l'interface du module. Donc un nouveau module. Par exemple si dans un système robotique on remplace un capteur odométrique basé sur un codeur incrémental (mesure du nombre de tours effectués par la roue d'un robot mobile) par une centrale inertielle (mesure des accélérations subies par le châssis), nous devons adapter tant les algorithmes qui assurent la communication avec l'appareil de mesure (protocole, données échangées, lien physique, etc.) que la traduction de ces mesures en valeurs de position. De plus, la centrale inertielle nous fournira

d'autres informations que le nouveau module pourra mettre à la disposition des autres (accélérations et vitesses instantanées par exemple).

Il est à noter que même si la modification affecte seulement l'algorithme et pas l'interface du module, on différenciera bien l'évolution d'un module induite par un changement matériel de l'évolution induite par l'amélioration d'un algorithme. En effet, dans le cas que nous venons d'évoquer, l'évolution devra donner lieu à un nouveau module adapté au nouveau matériel qui conférera au logiciel une rétro compatibilité puisque le retour à une ancienne configuration matérielle (retour à l'ancien capteur par exemple) se traduira au niveau de l'architecture par le remplacement du nouveau module par l'ancien.

Il est évident que le remplacement d'une instrumentation doit se faire de telle sorte que le changement apporte au minimum les fonctionnalités requises par le logiciel (possibilité d'extraire les mêmes types de données d'un capteur, possibilité d'induire un même mouvement avec un nouvel actionneur). L'interface du module gérant cette nouvelle instrumentation présentera des ports pouvant produire des mêmes variables que l'ancien module et, si la répartition des variables sur les ports a été faite de façon correcte (voir section 5.2.2), l'ajout du nouveau module à l'architecture ne nécessitera aucune modification des autres. On devra toutefois rediriger les nouveaux flux de communication avec le module ajouté, cela relève de l'évolution structurelle, pour partie automatiquement gérée.

L'architecture doit faciliter l'évolution structurelle qui s'exprime par le réagencement des modules selon un schéma déterminé par la fonctionnalité que l'on veut obtenir. Le schéma de commande de la figure 5.3 a été implémenté sous la forme de trois modules et on retrouve l'équivalent à la figure 5.6. Parmi ces trois modules, un des modules récupère les données capteurs, le suivant calcule la commande et le dernier se charge de l'appliquer aux actionneurs. Ce schéma de commande peut être employé par le robot lorsqu'il essaie d'atteindre une position donnée. Si maintenant le robot doit suivre une trajectoire définie par des éléments de son environnement (longer un mur par exemple), le schéma de commande précèdent devra changer.

5.4 Réutilisabilité

Certains modules développés pour un logiciel de contrôle donné doivent pouvoir être réutilisés dans un autre. Pour assurer cela il faut que les interfaces des modules suivent un certain "standard" de conception. D'une part il faut que les protocoles de transfert de données entre les ports des modules soient identiques (envoi de la donnée plus une date de production par exemple etc.), d'autre part il faut que le format de ces données ainsi que leurs unités soient identiques (cf section 5.4.4).

5.4.1 Au niveau de la conception

Comme illustré à la figure 5.5, un module est constitué d'une partie dédiée à son activité (calcul d'une loi de commande par exemple) et d'une autre partie dédiée à la gestion de son activité (initialisation, récupération et envoi des données, marche/arrêt etc.). La première partie, spécifique à la fonction que doit assurer le module, est à la charge du développeur. La seconde partie regroupe des mécanismes communs à tous les modules. Cette seconde partie sera réutilisée par le développeur lors de la conception

de son module. Il devra la compléter avec les paramètres relatifs aux flux de données (déclarations des variables à échanger par exemple).

La réutilisabilité peut aussi se faire à l'échelle d'un groupement de module accomplissant une fonctionnalité précise. Par exemple le schéma de la loi de commande de la figure 5.3 se traduit sur l'architecture par trois modules. Maintenant si l'on désire effectuer une commande courante en robotique, telle que la commande référencée capteur qui consiste à calculer la consigne selon les données perçues de l'environnement (suivi d'une ligne par exemple), on pourra réutiliser la commande déjà conçue précédemment. L'opération va consister dans un premier temps à construire le module capteur qui va permettre de mesurer la grandeur physique prise comme référence (position d'une ligne sur une image, valeur de température etc.) et le module qui va construire la consigne à partir de cette mesure, et dans un second temps à connecter ces modules à ceux qui implémentent la commande.

5.4.2 Au niveau de l'implémentation

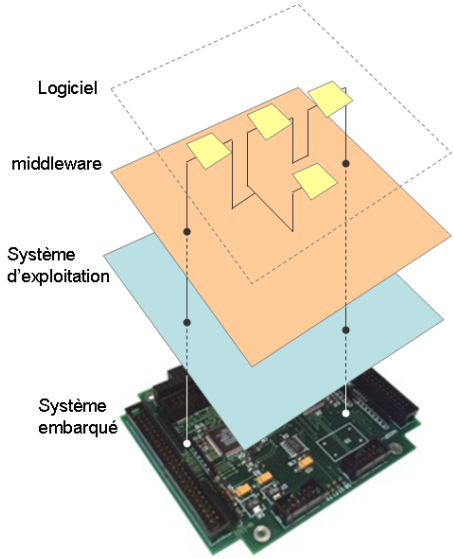


Figure 5.7 – Organisation des éléments logiciels et matériels d'un contrôleur

La figure 5.7 illustre les différents éléments matériels et logiciels d'un exemple de contrôleur. C'est cette structure que l'on rencontre le plus souvent dans notre contexte, à savoir un ordinateur embarqué relié aux capteurs et actionneurs de l'engin, avec un système d'exploitation sur lequel s'exécute le logiciel de contrôle. Le système d'exploitation offre une première abstraction du système matériel, il permet de fournir des mécanismes de communication entre éléments logiciels ou avec le matériel générique, quel que soit le système matériel qui le supporte.

Dans notre approche nous ajoutons un *middleware* qui sera une seconde abstraction

vis à vis du système d'exploitation. Ce *middleware* présente une API unique quel que soit le système d'exploitation qui le supporte.

Enfin, dans la couche la plus haute, on retrouve notre logiciel construit en modules. Le logiciel conçu pour une machine donnée peut être réutilisé dans une autre sans modification s'il s'agit du même système d'exploitation et moyennant le portage du *middleware* dans le cas d'un autre système d'exploitation.

De plus le *middleware* fait office de réceptacle commun aux différents acteurs (différents programmeurs), en masquant l'aspect système. Il offre un ensemble de primitives que l'utilisateur devra utiliser et qui permettront de rendre chaque implémentation de module conforme à des règles données (notamment les règles de communication et d'exécution).

5.4.3 Au niveau des communications

A la section 5.2.2, nous avons décrit comment les modules présentent leurs données pour les communications. On va maintenant décrire les mécanismes de communication que l'on va mettre en oeuvre pour faciliter la réutilisation des modules d'un logiciel de contrôle à l'autre.

Les données échangées au sein d'une architecture logicielle sont caractérisées par leur nature (par exemple vitesse, température, position), leur unité (mètre, degré, radian, etc.) et la taille de leur empreinte mémoire.

De manière générale, un module récupère des variables issues d'autres modules et met à son tour, certaines variables à disposition des autres modules de l'architecture. Exception faite des modules qui produisent des variables sans en consommer (modules capteurs généralement) ou qui consomment des variables sans en produire (modules actionneurs). Nous concevons ces communications inter modules selon le modèle **producteur-consommateur** avec un échange fondé sur la relation **publieur/souscripteur**[37]. Ce modèle suppose entre autre que les entités logicielles concernées (ici les modules) transfèrent leurs messages via une zone tampon (recopie du message vers le destinataire). Dans notre approche, pour chaque type de variable (au sens nature de la variable : température, vitesse etc.) produite ou consommée, nous associons un port sur le module. Ainsi dans un module, nous avons des ports de production et des ports de consommation.

La zone tampon requise par le modèle producteur-consommateur sera créée au niveau des ports de consommation et aura une taille multiple de la taille des variables échangées. Il est à noter qu'on peut échanger entre deux ports une seule variable ou bien un ensemble de variables (voir section 5.2.2). Lorsqu'il s'agit d'un ensemble de variables, l'échange se fait en une seule fois à l'instar de l'échange d'une variable unique. Nous nommerons "postage" l'action de transférer une variable d'un module vers un autre.

Le module producteur est en charge du postage, c'est à dire qu'il doit recopier les variables qu'il a produites dans les zones tampons des ports des modules consommateurs. Pour pouvoir poster ses données produites, le module producteur doit connaître la liste des destinataires pour chacune de ses données. Or la liste des destinataires (i.e. des consommateurs) peut évoluer dans le temps, selon le contexte. Pour cela nous avons mis en place un mécanisme d'abonnement du consommateur au producteur pour une variable donnée. Ainsi le module consommateur d'une variable donnée va s'abonner auprès du module producteur de cette variable en lui précisant le port de production (pour désigner la variable désirée parmi celles proposées par le producteur) et le port de consomma-

tion : il souscrit (s'abonne) à la publication de cette variable. On établit ainsi un lien de communication qui est caractérisé par quatre informations : module producteur, port de production, module consommateur, port de consommation.

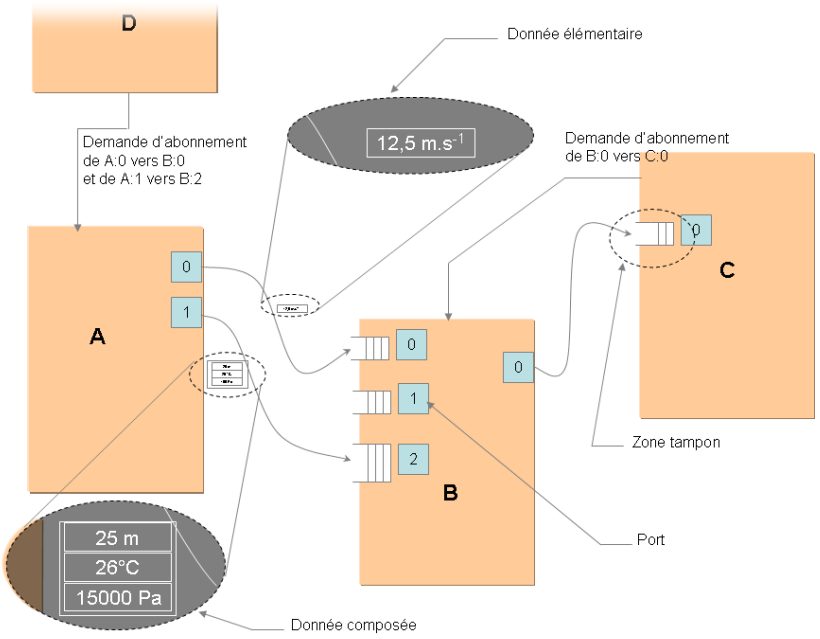


Figure 5.8 – Echange de données entre modules

Lorsqu'un lien de communication a été établi entre deux ports de deux modules, le module producteur envoie à chacune de ses exécutions (ou selon un multiple de cycles indiqué par le consommateur) la variable qu'il a produite au module consommateur. Pour rompre ce lien, il suffit au module abonné de se désabonner auprès du module producteur. Pour l'établir (ou le rétablir) il suffit de s'abonner (se ré-abonner).

Lorsqu'on construit un module pour une application précise (schéma de commande), on peut y inclure les déclarations des abonnements qu'il doit effectuer afin d'établir les liens de communication qui vont lui fournir les données dont il aura besoin pendant son exécution. Cependant certains modules effectuent des tâches assez génériques pour pouvoir être réutilisés soit dans un autre schéma de commande soit dans une autre architecture logicielle. Pour éviter d'intervenir au sein du module et modifier la liste des abonnements qu'il doit effectuer, il est possible de confier l'abonnement à une entité extérieure au module. Cet élément logiciel que nous expliciterons plus tard, pourra dynamiquement établir et rompre n'importe quel lien de communication entre deux modules.

Nous distinguons alors deux cas dans l'établissement des liens de communication entre modules. Le cas où la consommation du module est décrite en son sein et effectue lui-même sa demande d'abonnement (auprès des modules producteurs concernés) et le cas où un module tiers abonne un module à un autre. Dans le premier cas, on parlera d'abonnement "statique" car celui-ci est décrit dans le module et ne pourra être modifié que si on modifie le module. Dans le second cas, on parlera d'abonnement "dynamique" puisque celui-ci étant fait par un module tiers, il pourra être modifié à tout moment. Notons que

Table 5.1 – Unités de base du Système International (SI)

Unité	*Grandeur*	*Symbole*
Le mètre	Longueur	m
Le kilogramme	Masse	kg
La seconde	Temps	s
L'ampère	Courant électrique	A
Le kelvin	Temperature	K
La mole	Quantité de matière	mol
La candela	Intensité lumineuse	cd

tous les échanges sont automatiquement datés (de manière transparente pour l'utilisateur, le programmeur).

5.4.4 Le besoin de standardisation

La réutilisation suppose :
- compatibilité fonctionnelle,
- compatibilité des formats des données,
- compatibilité des unités de données.

On désigne par compatibilité fonctionnelle, le fait que le module réutilisé remplisse une fonction nécessaire dans le contexte de réutilisation. Par exemple si on a besoin d'un module qui effectue le calcul d'un modèle géométrique il est évident que si un module ne remplit pas au moins cette fonctionnalité, il ne peut être réutilisé pour ce cas.

La compatibilité des formats des données concerne la façon dont une donnée va être encodée. Sur la plupart des langages de programmation on trouve des formats élémentaires prédéfinis (int, double, float en langage C par exemple) ainsi que des formats utilisateurs construits par composition des formats élémentaires. Il va de soi que deux modules qui s'échangent une donnée doivent connaître le format de la donnée afin de pouvoir l'interpréter correctement.

Pour réaliser leurs tâches, les systèmes robotiques sont amenés à mesurer les grandeurs du système physique qui les constitue (position, vitesse etc.) ainsi que du milieu qui les entoure (température, position d'obstacles etc.). Toutes ces mesures vont être constituées d'une valeur numérique ainsi que d'une unité. Une unité va servir à caractériser la valeur numérique d'une grandeur physique. Un système d'unités est un ensemble de règles qui définit de manière cohérente l'unité de mesure de chaque grandeur utilisée dans les sciences et les techniques. Le système d'unités qui est aujourd'hui utilisé dans le monde entier est le Système international d'unités (SI). Il a été introduit lors de la 11e Conférence Générale des Poids et Mesures (CGPM) en 1960. On distingue deux classes d'unités dans le Système International : les unités de base et les unités dérivées. Les unités de base sont répertoriées dans le tableau 5.1.

Les unités dérivées sont formées à partir des unités de base en utilisant les mêmes relations algébriques que celles s'appliquant aux grandeurs correspondantes sur la base des lois physiques. Un point important est la cohérence du système international d'unités, c'est à dire que les unités dérivées s'obtiennent par multiplication et division d'unités de base, sans faire intervenir de facteurs numériques supplémentaires.

Il est nécessaire d'assurer la compatibilité des unités des données échangées entre les modules. Un module conçu pour produire une donnée avec un certain format et une unité donnée ne peut être relié (au sens de la composition) qu'à un (ou plusieurs) module consommant une donnée avec les mêmes caractéristiques.

Pour une architecture logicielle considérée, il est facile de poser des conventions sur les caractéristiques des données à échanger entre les modules. Si on veut réutiliser un module dans une autre architecture logicielle, on doit veiller à ce qu'il respecte les conventions du logiciel d'accueil. La réutilisabilité se retrouve en ce point limitée par les différentes conventions établies pour chaque projet d'architecture logicielle.

Pour éviter ces problèmes de compatibilité, plusieurs groupes de travaux scientifiques, (JAUS [5], OMG) essayent d'établir un standard sur les formats (empreinte mémoire, unité etc.) d'échange de données dans le domaine des logiciels pour la robotique. Ces efforts de standardisation sont rendus nécessaires par l'apparition de nombreux systèmes robotiques manquant d'interopérabilité. Ils vont traiter tant des aspects internes au logiciel (standardisation des unités de mesures) que les aspects lié à la coopération entre plusieurs logiciels (standardisation des protocoles de communication). Cet aspect ne sera pas approfondi dans la thèse.

5.5 "Contrôlabilité" et réactivité

5.5.1 Contrôle de l'activité

Un logiciel de contrôle robotique enregistre un ensemble de mesures capteurs et, au regard de sa mission, va produire un ensemble de commandes pour les actionneurs. Pour cela, il agrège un ensemble de traitements que nous avons répartis dans des entités appelées "modules". La réalisation des fonctionnalités de contrôle d'un système robotique, nécessite l'activation d'un ensemble de modules, et ce dans un ordre précis. En effet, ces modules vont échanger des données entre eux et pour assurer la cohérence des traitements, ils devront être exécutés dans un ordre précis. Un module A produisant une variable x consommée par un module B, doit avoir terminé son exécution avant que B ne commence à s'exécuter.

Pour garantir les relations de précédence entre les modules, relations contextuelles (i.e. évoluant dans le temps), on doit déporter la décision d'exécution des modules à l'extérieur de ceux-ci. La relation de précédente entre modules va demander la connaissance des différents modules qui vont entrer en jeu dans la fonctionnalité de contrôle considérée. Cette connaissance devrait être présente dans chaque module si la décision du moment d'exécution était prise au sein du module. Cela aurait pour conséquence de briser la modularité puisque pour employer un module donné dans une autre fonctionnalité de contrôle, il faudrait modifier la relation de précédence au sein du module. Or il est essentiel de préserver la possibilité de composition "dynamique" (en opposition à statique, i.e. figée) des modules.

Donc, cette décision d'exécution sera prise par un autre élément logiciel spécifique, unique, relevant de l'infrastructure d'exécution des modules, que l'on appellera "ordonnanceur" et dont on détaillera le fonctionnement au chapitre suivant. Dans ce cas, chaque module devra alors attendre une requête d'activation pour s'exécuter. Son exécution devra être non cyclique et d'une durée bornée. A la fin de son exécution, il devra de nouveau retourner en attente d'une nouvelle requête.

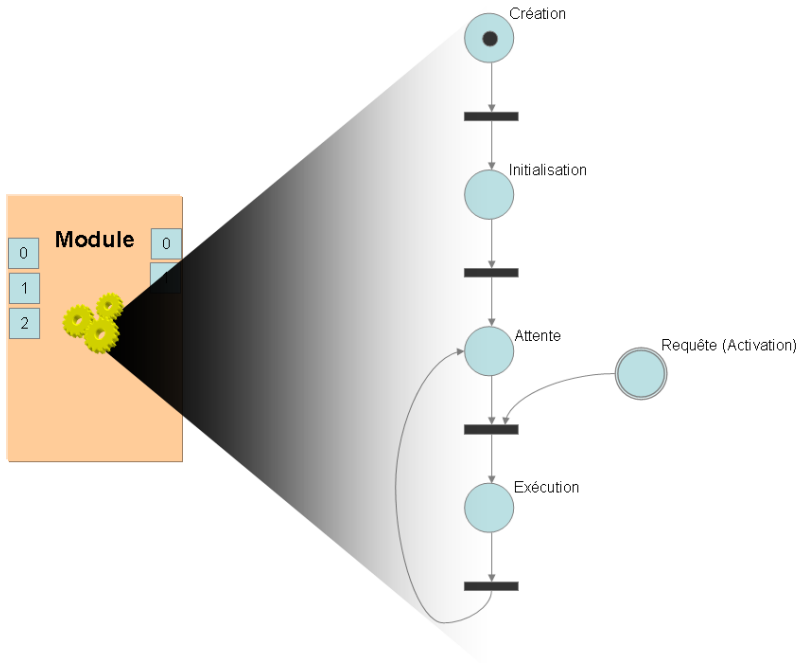


Figure 5.9 – Modèle simplifié (basé réseau de Petri) d'un module

Nous parlons de "contrôlabilité" du module puisque son activité est contrôlée de façon externe. Ce contrôle peut être plus ou moins fin, des classiques activation/suspension/arrêt à la modification de ses paramètres internes (influençant potentiellement le déroulement de son exécution).

La séquence de traitement d'un module (un module ne comprenant pas de parallélisme) est découpée en un ensemble de pas, i.e. d'étapes ou de phases. Le passage d'une phase à une autre se fera que si aucun ordre d'arrêt (ou de suspension) n'a été reçu par le module. A chacune de ces phases, le module peut être interrompu. La décomposition en phase fixe donc la réactivité du module.

Le degré de réactivité d'un module va se mesurer au temps qu'il va mettre pour réagir à une requête. Il est évident ici que ce temps va être fonction des mécanismes de communication ainsi que de la durée de chaque phase. Les mécanismes de communication asynchrones mis en jeu dans l'envoi d'une requête auront une durée bornée et sensiblement identique pour tous les modules. Le degré de réactivité d'un module va alors reposer sur la granularité de son exécution. Une forte granularité engendrera un haut degré de réactivité et une faible granularité une forte latence.

Il faut en effet établir un compromis en terme de granularité, i.e. décomposition en phases (étapes) de la séquence d'exécution du module : il ne s'agit pas de stopper un

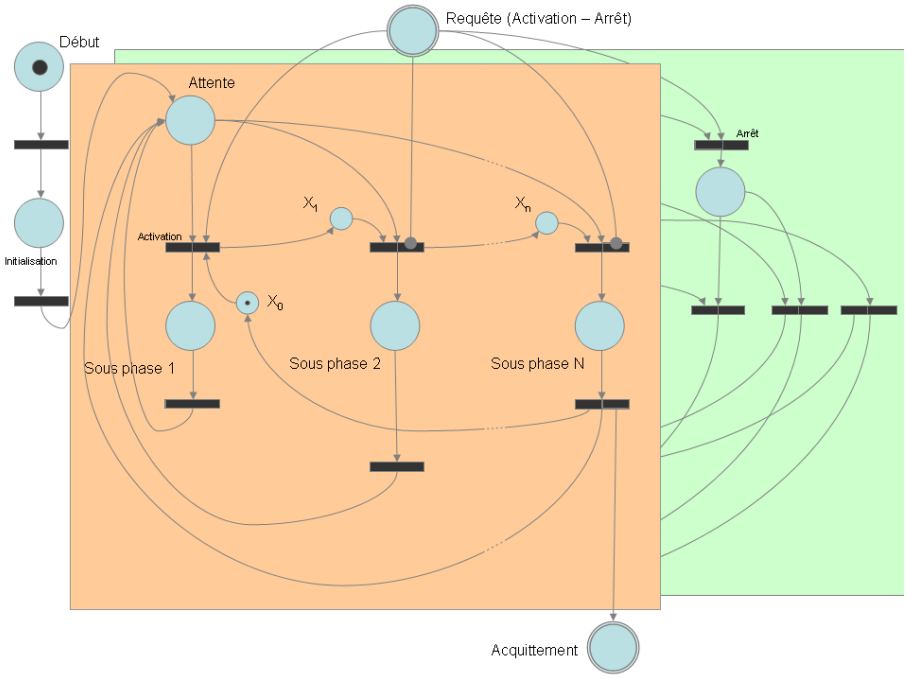


Figure 5.10 – Modèle simplifié (basé réseau de Petri) de la gestion des requêtes d'arrêt au sein d'un module

processus (au sens système) mais bien de suspendre l'exécution d'un module dans un état connu, i.e. à partir duquel une reprise est possible. La prise en compte des requêtes externes se faisant entre chaque phase, l'impact de la granularité de la décomposition est évident : à gros grain la latence de réaction est plus conséquente, à grain fin l'exécution du module est pénalisée.

Pour activer un module, on doit vérifier au préalable que le module qui le précède ait terminé son exécution. Pour déterminer la fin de l'exécution d'un module, ce dernier renvoie un acquittement. Cet acquittement permet en outre de surveiller le bon déroulement de l'exécution et de faire face aux situations anormales (ex : exécution trop longue).

5.5.2 Interactions

Lorsqu'un module reçoit une requête d'activation, il déroule son algorithme dans lequel il utilise les données reçues sur ses ports d'entrée et en produit à son tour sur ses ports de sortie. La consommation et la production de données par un module s'effectuent durant son exécution. Les échanges de données sont alors systématiques entre les modules. Il s'agit là d'une méthode de communication inter module qui est requise dans bon nombre de cas, dont le cas courant d'une implémentation d'un schéma de commande tel qu'illustré à la figure 5.3. Sur cette figure, un schéma de commande est implémenté par trois modules : un module capteur, un module loi de commande et un module actionneur.

Le module capteur doit être activé en premier, doit produire sa mesure, puis le module loi de commande doit être activé, il doit consommer la mesure du module capteur et doit produire à son tour la commande, enfin le module actionneur qui doit être activé en dernier, doit consommer la commande et l'appliquer aux capteurs. Cette série d'activations

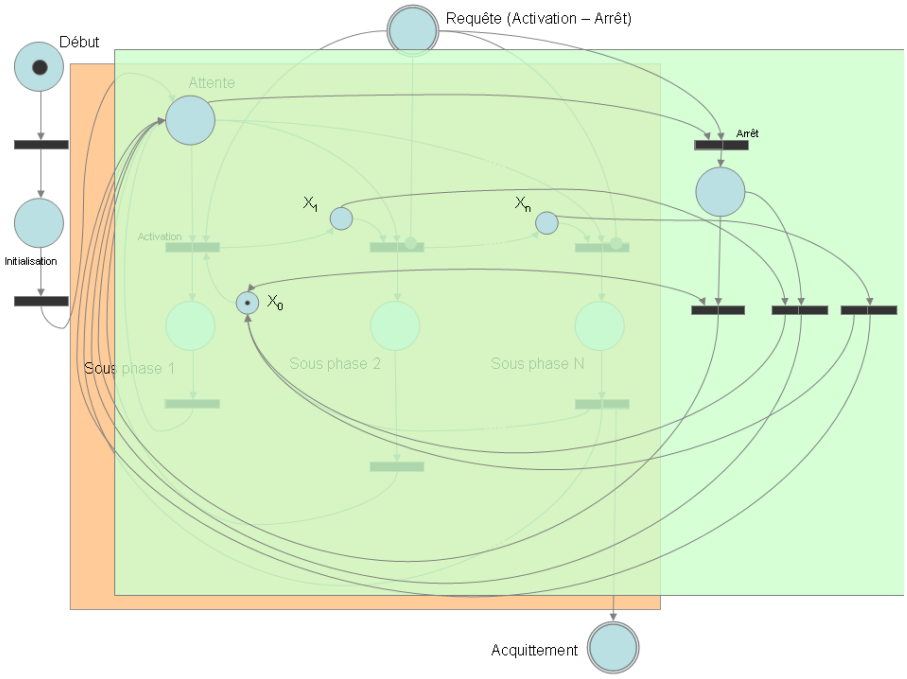


Figure 5.11 – Modèle simplifié (basé réseau de Petri) de l'enchaînement des phases au sein d'un module

et d'échanges de données doit toujours se faire dans le même ordre (pour un contexte donné) à une périodicité choisie, au regard notamment de considérations de stabilité de la commande.

En revanche nous pouvons avoir besoin dans une architecture logicielle, d'un autre type d'échange de données. Pour illustrer cela, nous considérerons le cas d'un véhicule sous-marin qui utilise une loi de commande pour se déplacer dans l'eau peu profonde dont les gains doivent être modifiés dès que le véhicule dépasse une certaine profondeur. Ce schéma de commande donnera lieu à la création de quatre modules A,B,C et D :

- A : module capteur qui récupère la position du véhicule à partir de l'instrumentation (centrale inertielle par exemple),
- B : module commande qui implémente la loi de commande et un comparateur,
- C : module actionneur pour appliquer les commandes à l'engin,
- D : module capteur qui va mesurer la pression de l'eau.

L'échange de données entre les modules A et B d'une part et les modules B et C d'autre part seront systématiques. C'est à dire que si ces modules sont activés périodiquement comme l'imposent les règles de l'automatique, alors les échanges évoqués précédemment seront périodiques. Par contre les échanges de données entre le module D (mesure de pression) et le module B (commande) n'ont pas besoin d'être systématiques. En effet, la commande n'a pas besoin de la mesure instantanée de pression, elle a seulement besoin de connaître l'information "seuil de pression dépassé". On appellera ce type d'information, un événement.

Dans notre approche, un événement est constitué d'une variable qui peut contenir la valeur de la variable surveillée. Comme illustré à la figure 5.13, un port est affecté à chaque événement produit par un module et un autre pour chaque événement consommé par le

module. Tout module sensible à un événement doit s'abonner auprès de son générateur (i.e. producteur). L'événement n'est notifié qu'une seule fois (lors de son occurrence) à tous les abonnés : le module producteur de l'événement résilie l'abonnement du module consommateur après la production de l'événement. Cela est justifié par le fait que si l'événement résulte de la surveillance d'une variable, toute la durée où la valeur de cette variable aura dépassé le seuil considéré, le module ne cessera de produire des événements induisant un flot de messages inutiles, à éviter d'autant plus qu'ils peuvent engendrer des commutations de processus inutiles également.

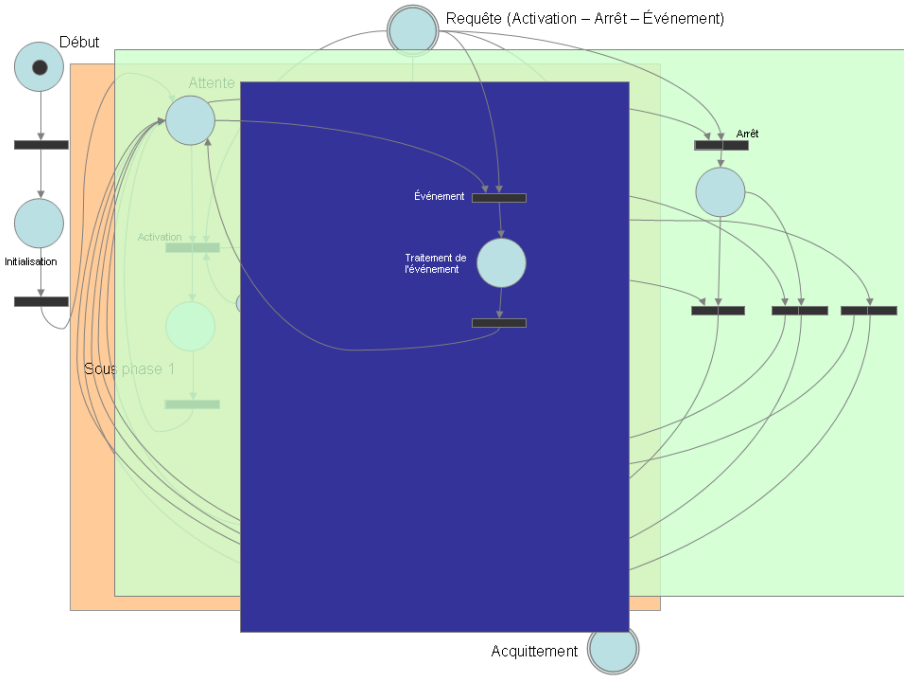


Figure 5.12 – Réseau de Petri d'un module : gestion des événements (reçus)

Un événement, contrairement à une donnée peut induire la modification de la séquence de traitements effectuée à un instant donné par un module. Si le module de commande de l'exemple évoqué précédemment reçoit un événement du module de mesure de pression, il devra modifier la valeur de ses gains, au plus vite, avant le calcul de la commande. Pour cela un événement reçu par un module va interrompre les traitements en cours (voir figure 5.12).

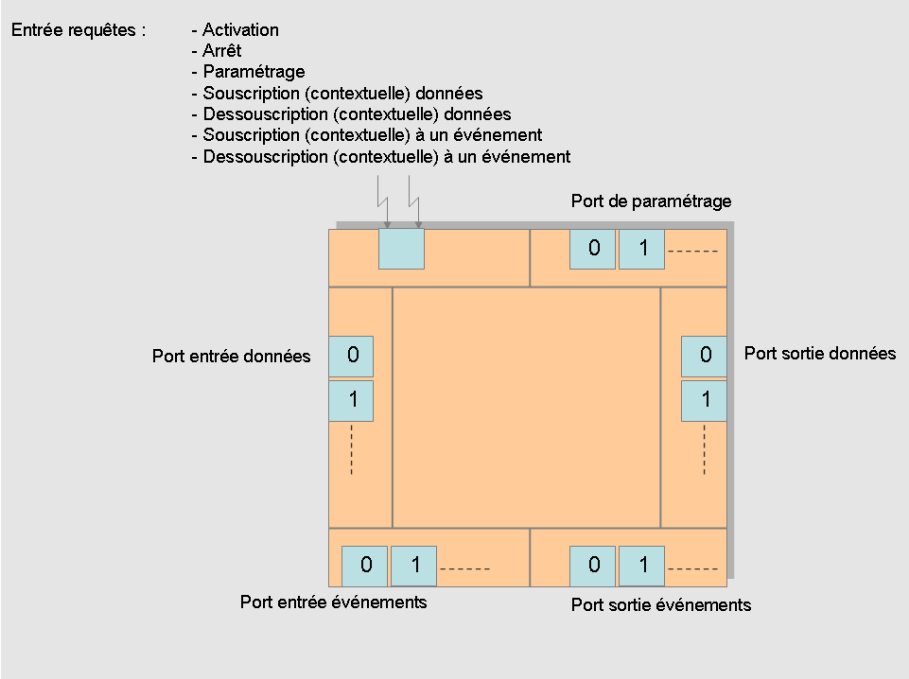


Figure 5.13 – Récapitulatif des ports d'un module

Résumé du chapitre 5 :

- Un système robotique se compose d'une partie contrôle et d'une partie opérative. Cette partie contrôle nommée contrôleur est généralement centralisée autour d'un logiciel de contrôle, pour les robots tels que nos AUV.
- L'établissement d'un logiciel de contrôle passe par la construction d'une architecture logicielle qui devra respecter certains critères de conception : modularité, évolutivité et réutilisabilité étant des propriétés fondamentales.
- Notre architecture logicielle est construite par assemblage d'éléments de base appelés "modules".
- L'assemblage de plusieurs modules consiste à établir des liens (dynamiques ou statiques) d'interaction entre les ports des différents modules. Ces liens vont être les supports des flux de données et de contrôle au sein de l'architecture.
- Un module présente six catégories différentes de ports :
  - ports d'entrée de données
  - ports de sortie de données
  - ports d'entrée d'événement
  - ports de sortie d'événement
  - ports de paramétrage
  - port d'entrée des requêtes
- L'activité des modules est contrôlée par un module particulier appelé "ordonnanceur"

Chapitre 6

Construction d'une architecture logicielle

6.1 Introduction

Dans le domaine de la robotique, et plus précisément en robotique mobile, le véhicule est plongé dans un milieu physique souvent inconnu ou mal connu (dont la structure évolue au cours du temps : couloirs d'un immeuble, fonds marins etc.). Dans ces conditions, l'homme ne peut indiquer, a priori, au robot toutes les caractéristiques du milieu, nécessaires au bon accomplissement de sa mission. Le robot devra alors se contenter d'une mission décrivant les principales tâches à effectuer et doit se charger lui-même d'adapter son action aux exigences de l'environnement.

On distingue alors deux aspects pour la prise de décision du robot (logiciel de contrôle) : l'évaluation de l'intention de l'opérateur (mission), l'examen des obligations imposées par l'environnement et le matériel. A partir de ces intentions et obligations, le contrôleur robotique est en charge des différentes prises de décision qui vont amener l'engin à accomplir sa mission. Dans le chapitre précédent nous avons étudié sur quelles bases un logiciel de contrôle robotique devait être architecturé pour respecter certaines propriétés, dont la modularité, l'évolutivité et la réutilisabilité. Cette étude nous a mené à décomposer le logiciel de contrôle en entités de base appelées "modules".

Nous allons voir dans ce chapitre comment assembler les entités de base "les modules" afin de construire un logiciel de contrôle robotique qui va permettre d'accomplir sa mission au regard des contraintes de son milieu. Dans une première section nous allons voir comment les modules sont hiérarchisés reflétant les différents niveaux de prise de décision au sein de l'architecture logicielle. Dans une seconde section nous étudierons la façon dont le contexte est pris en compte dans la décision. Enfin, dans une dernière section nous étudierons les mécanismes de gestion des événements dans l'architecture.

6.2 Répartition de la décision au sein du modèle d'architecture

L'intention est représentée par l'ordre donné par l'opérateur (plus ou moins directement, selon qu'il s'agisse de l'exécution d'une mission ou d'une téléopération) et l'obligation par l'ensemble des contraintes imposées par l'environnement et/ou par l'instrumentation de bord. L'intention doit alors se raffiner depuis son expression haut niveau (sous forme d'objectif) pour donner naissance à des commandes bas niveau (application périodique d'une loi de commande par exemple) appliquées à l'instrumentation. D'un autre coté, les contraintes bas niveau (obstacle, ressources en nombre limité, etc.) doivent être remontées et agrégées pour pouvoir être interprétées à un niveau d'abstraction plus élevé (la valeur d'un capteur de distance doit être transformée par exemple en l'événement "haut-niveau" obstacle à proximité). L'obligation rend compte des possibilités physiques du véhicule et l'intention de la volonté de l'utilisateur. Ces deux notions doivent être intégrées pour pouvoir d'une part mettre en oeuvre les actions robotiques voulues et d'autre part respecter les limitations physiques du véhicule ou les contraintes liées à son utilisation dans l'environnement.

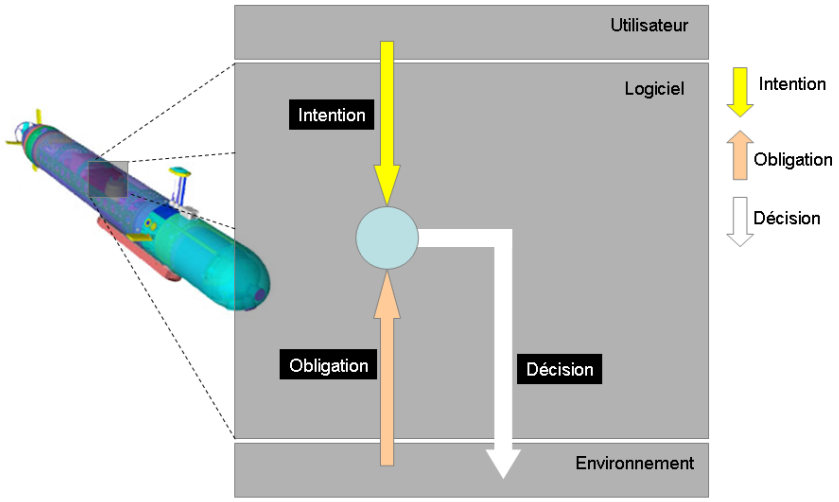


FIGURE 6.1 – Intégration de l'intention et de l'obligation pour la prise de décision

Pour respecter certains critères de conception, nous avons structuré le modèle de notre architecture en un ensemble de modules indépendants reliés par des liens de communication (voir chapitre 5). Les liens de communication sont de deux sortes, les premiers supportent les flux de données (ou d'événements), les seconds correspondent aux flux de contrôle des modules. On fait alors apparaître une hiérarchie entre les modules, traduite par les liens de contrôle et par la disposition relative de ces modules dans le modèle d'architecture. Les modules qui exercent un contrôle sont situés au dessus de ceux qui

subissent ce contrôle. Cette hiérarchisation classique va être au coeur de la répartition de la décision au sein du modèle d'architecture. Au plus bas niveau de cette hiérarchie, on retrouve l'ensemble des modules en relation avec l'instrumentation appliquant les décisions des niveaux supérieurs. Au plus haut niveau, les modules gérant la mission, capable d'en modifier le cours ou de l'arrêter.

En analysant les intentions et les obligations, nous pouvons identifier la répartition de la décision dans les différentes strates du modèle d'architecture. L'opérateur exprime son intention au robot sous la forme d'une mission, c'est-à-dire d'une suite d'objectifs à mener à bien. Cette intention est prise en compte dans le plus haut niveau du modèle d'architecture et doit donner lieu, par raffinements successifs (contrôle de l'activité des modules d'un niveau inférieur), à la commande périodique des actionneurs qui amènent le robot à effectuer l'action correspondante.

Notons que l'approche modulaire que nous proposons permettrait tout aussi bien de structurer les modules différemment, selon une architecture comportementale par exemple (cf chapitre 2).

6.2.1 Superviseur Global

Recevant directement les intentions de l'opérateur sous la forme d'une mission, un module situé dans le haut niveau que nous appelons, **superviseur global (SG)**, doit prendre les décisions relatives à cette mission puis les transmettre aux modules des niveaux inférieurs.

Les décisions alors transmises par le SG au niveau inférieur seront appelées des "Objectifs". Une base de données (voir figure 6.2) consultée par le SG lui permet de connaître en quels objectifs la mission reçue doit être décomposée.

Les obligations à prendre en compte à ce niveau là sont de nature à influencer le cours de la mission comme le report, l'annulation d'un objectif ou l'arrêt complet de la mission. Ces obligations notifiées de manière événementielle peuvent concerner par exemple la panne d'un appareil, le passage du système dans une zone dangereuse, la détection d'une entrée d'eau, etc.... Ces obligations sont convoyées par des flux de données ascendant qui prennent source dans les modules de bas niveau (en relation avec l'instrumentation de bord) et à destination des modules haut niveau. Dans un schéma de hiérarchisation pur, ces flux doivent traverser les différentes couches jusqu'à atteindre la couche concernée. Comme évoqué dans l'état de l'art, cette approche réduit la réactivité de l'architecture. Pour assurer la réactivité, les flux de données traduisant une obligation seront alors directement acheminés sous forme d'événement (lien direct entre deux modules quels que soient leurs niveaux respectifs dans le modèle d'architecture) vers les modules concernés.

D'autres obligations prises en compte au niveau du SG ne sont pas issues d'une interrogation de l'instrumentation mais traduisent les capacités et les disponibilités du matériel. Ces obligations sont des connaissances du niveau supérieur sur les propriétés de l'engin ainsi que son état. Pour pouvoir appliquer des règles sur l'état de ce matériel, on le modélise sous forme de ressources et de macro-ressources. Une ressource est un matériel qui peut avoir un ensemble de fonctionnalités propre indépendamment du reste (caméra, etc.). Une macro-ressource est issue de la combinaison d'un ensemble de ressources en un tout relativement indépendant. Une macro-ressource doit avoir une fonctionnalité propre et peut être utilisée seule pour accomplir une tâche robotique particulière parmi celles pour

lesquelles l'engin a été construit. Par exemple, le bras et la base mobile du manipulateur mobile sont tous deux des macro-ressources. En effet la tâche robotique de déplacement, ne requiert que la base mobile et la tâche de préhension ne requiert que le bras articulé.

L'intention de l'utilisateur exprimée sous la forme d'une mission et les obligations issues du matériel et de l'environnement constituent la base sur laquelle le SG fonde sa décision. Cette décision sera par la suite transmise aux niveaux inférieurs pour lesquels elle constituera une intention qu'on appellera "Objectif".

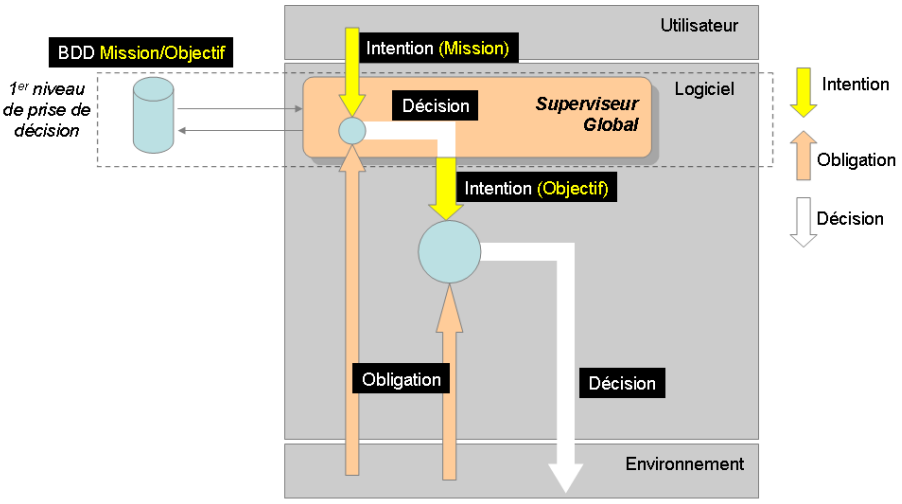


Figure 6.2 – Premier niveau de prise de décision, le superviseur global

6.2.2 Superviseur Local

Dans ce niveau on dispose d'un ensemble de macro-ressources et d'un ensemble d'objectifs.

Une macro-ressource est le fruit du regroupement de plusieurs ressources. Par exemple, nous pouvons regrouper tous les éléments nécessaires à la manipulation dans une macro-ressource **bras**, tous les éléments nécessaires à la locomotion dans une macro-ressource **base mobile**. Il est à noté qu'il est également possible de percevoir le robot dans son ensemble comme une macro-ressource (auquel cas l'ensemble de ses degrés de liberté peuvent être impliqués dans une même loi de commande par exemple).

Ces macro-ressources, bras et base mobile par exemple, peuvent être pilotées/contrôlées selon des modes différents et ce simultanément. En effet on peut distinguer trois modes de fonctionnement : autonome, téléopéré (Homme-Machine), coopératif (Robot-Robot). Dès lors il est possible d'engager des macro-ressources dans des modes différents : par exemple nous pouvons laisser opérer le bras dans le mode autonome pendant que la base mobile est téléopérée (ex : transport de charge explosive, compensation automatique des

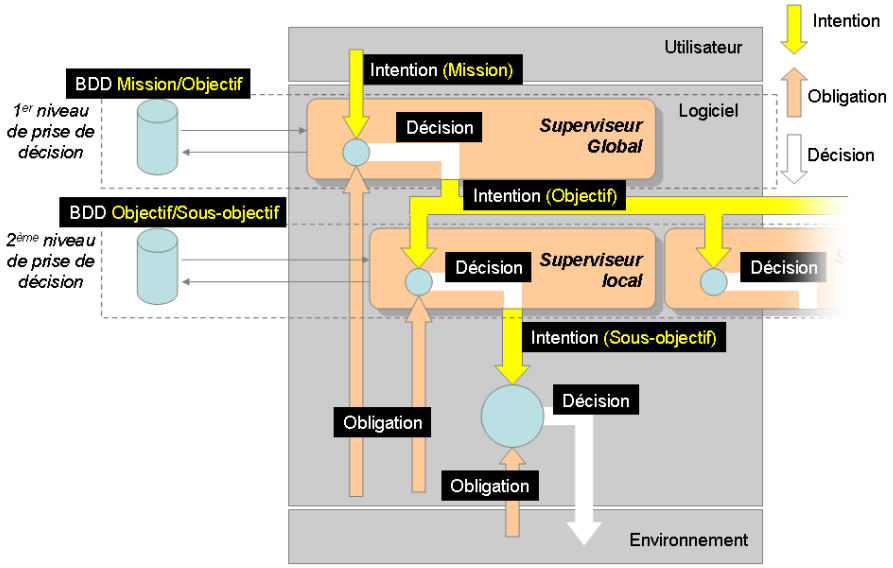


Figure 6.3 – Deuxième niveau de prise de décision, les superviseurs locaux

perturbations sur le bras par le mode autonome alors que l'opérateur téléopére la base mobile pour conduire le robot en un point précis).

On définit donc des couples {macro-ressource, mode}. Pour chacun des couples nous construirons un superviseur local.

Dans notre cas, nous disposons que d'une seule macro-ressource, le véhicule, avec pour l'instant deux modes :

- autonome (mission)
- téléopéré (uniquement en surface, par exemple pour ramener le véhicule au bateau)

Cependant nous pourrions aisément évoluer si on ajoutait un "bras" au véhicule.

Ces modes de fonctionnement sont exclusifs entre eux et doivent répondre à des contraintes et surveiller des événements de nature différente : par exemple, la surveillance de l'interruption de la liaison avec l'opérateur est pertinente en mode téléopération mais pas en mode autonome. De ce fait on attribuera une entité informatique particulière pour gérer chaque couple {macro-ressource, mode} utilisable sur le système : cette entité est un **Superviseur Local**. Ce découpage est à la fois nécessaire pour la clarté du code implémenté et répond aussi à un besoin de modularité et d'évolutivité (l'ajout d'un nouveau mode de fonctionnement n'affecte pas les précédents, sous réserve d'assurer l'exclusion mutuelle pour une même macro-ressource).

Le choix du SL à activer à un moment donné revient au SG. Lorsqu'un SL a été choisi, il sera en charge de mener à bien l'objectif que lui confiera le SG. Un objectif constitue une étape élémentaire de la mission. Cette étape est élémentaire du point de vue de l'utilisateur (elle correspond à une tâche robotique particulière) mais peut nécessiter plusieurs phases dans sa réalisation. Ces phases sont distinguées par l'ensemble de matériels et de traitement employés. Par exemple l'objectif suivre un pipeline pour un vé-

hicule sous-marin, nécessite le passage par les phases suivantes : plonger, naviguer jusqu'à la position du pipeline, rechercher le pipeline, suivre le pipeline. Ces phases sont appelées sous-objectifs.

Pour réaliser un objectif donné, le SL devra alors le décomposer en plusieurs sous-objectifs qu'il transmettra au niveau inférieur. Les sous-objectifs nécessaires à la réalisation d'un objectif sont contenus dans une base de données (voir figure 6.2) consultée par le superviseur local.

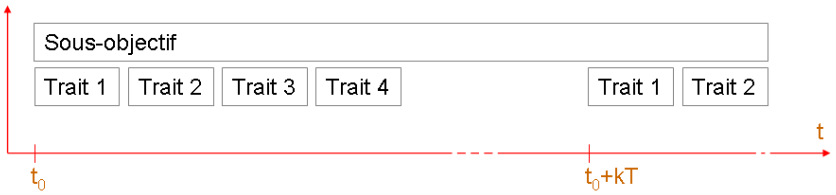


Figure 6.4 – Composition d'un sous-objectif (ensemble de traitements représentant l'exécution de modules)

Un sous-objectif correspond à une phase homogène, dans le sens où l'ensemble des moyens d'action et de perception qui vont être mis en oeuvre durant cette phase seront les mêmes du début jusqu'à la fin[38]. La suite de traitements qui va permettre d'exécuter un sous-objectif, est construite sur un ensemble de motifs potentiellement répétés à une période de temps fixée (voir figure 6.4). Dans la plupart des cas (notamment dans le cadre de la commande), le premier traitement sera l'interrogation des capteurs, le deuxième le traitement des données, le troisième, le calcul de la commande et enfin le quatrième, l'émission des commandes vers les actionneurs.

6.2.3 Exécutif

L'exécutif regroupe l'ensemble des modules bas niveau et l'ordonnanceur. Ce niveau est en charge de la réalisation effective des décisions haut niveau. Ces décisions sont exprimées sous la forme de sous-objectifs. L'exécution d'un sous-objectif par le bas niveau se traduit par la mise en oeuvre d'un certain nombre de traitements informatiques qui vont de l'interrogation des capteurs à l'application des commandes aux actionneurs.

Chaque traitement est implémenté dans un module spécifique situé au bas niveau de l'architecture. On distingue quatre catégories de modules bas niveau. Les lois de commande sont dans les Modules d'Action, les différents traitements effectués sur les données perçues sont dans les Modules de Perception, enfin la commande des actionneurs et la configuration des capteurs sont respectivement dans les Modules Actionneurs et les Modules Capteurs.

Exécuter un sous-objectif issu d'un SL, revient à exécuter les traitements qui le composent. Un traitement donné est lancé lorsque le module qui le contient est activé. Cette activation prend effet lorsque le module concerné reçoit un ordre d'activation. Ces différents traitements ont une durée bien précise.

Lorsque plusieurs sous-objectifs sont amenés à être exécutés en même temps comme le montre la figure 6.5, leurs traitements se retrouveraient superposés dans le temps.

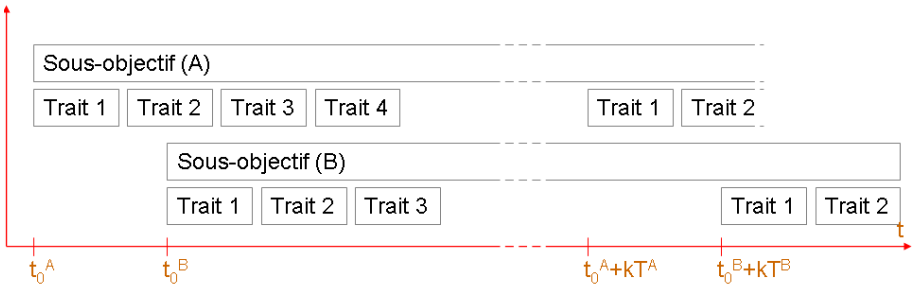


Figure 6.5 – Exécution de deux sous-objectifs

Les architectures matérielles des véhicules d'intervention que nous considérons, les petits AUVs en l'occurrence, ne possédant, dans la plupart des cas, qu'une seule unité de calcul, il devient alors nécessaire de pouvoir contrôler l'ordre d'exécution de ces traitements. D'autant plus que certains résultats, comme les commandes des actionneurs, doivent être fournis avant des dates précises : contexte temps-réel.

Exécutés sur une unique unité de calcul, ces traitements sont équivalents à des tâches informatiques en concurrence pour l'utilisation d'un processeur. Afin de déterminer l'ordre d'activation de chaque tâche en fonction des sous-objectifs à exécuter un module particulier appelé "ordonnanceur" reçoit les sous-objectifs des superviseurs locaux et se charge d'activer périodiquement les modules concernés. La liste des modules à activer selon une période donnée pour effectuer un sous-objectif est contenue dans une base de données consultée par l'ordonnanceur (voir section 6.5).

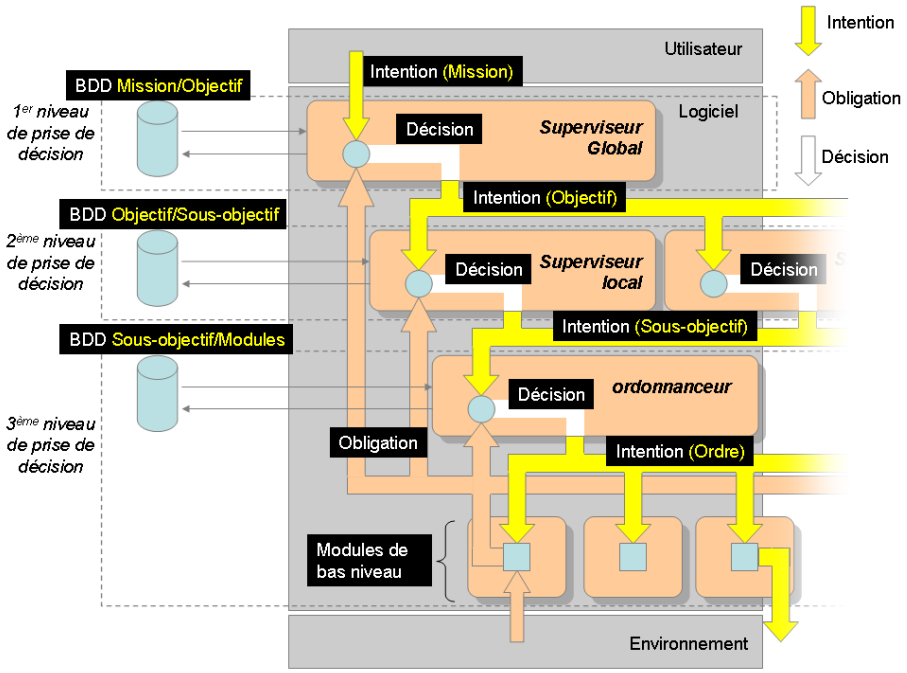


Figure 6.6 – Troisième niveau de prise de décision, l'ordonnanceur bas niveau

6.3 Gestion contextuelle de tâches

6.3.1 Mission décrite par un ensemble d'objectifs

Un des points importants dans la navigation autonome est la capacité de l'engin à prendre seul les décisions stratégiques relatives au déroulement de la mission. Nous pouvons étudier la mission à différent niveaux de granularité : un niveau très agrégé où l'on va raisonner sur les principales étapes de la mission comme l'inspection d'un pipeline ou la bathymétrie d'une zone, le très bas niveau où on déterminera périodiquement les commandes à appliquer aux actionneurs du véhicule et où on surveillera l'évolution des grandeurs physiques à mesurer, enfin un niveau intermédiaire où on décomposera les principales étapes de la mission en sous étapes élémentaires.

Afin de pouvoir appliquer des raisonnements sur ces niveaux, nous leur avons donné existence sous la forme d'un vocabulaire utilisé par le contrôleur et servant aussi d'interface de programmation avec l'utilisateur. Nous avons déjà introduit dans la section précédente l'alphabet et les mots de ce vocabulaire : les objectifs, les sous-objectifs et les modules.

Les objectifs sont manipulés par le superviseur global, les sous-objectifs par le superviseur local et les modules par le niveau exécutif.

La programmation se fait par la constitution d'une mission. Une mission est une suite d'objectifs à accomplir. Ces objectifs représentent alors les intentions de l'utilisateur.

D'un autre coté la capacité du véhicule est traduite par les modules bas niveau qui ont été construits pour gérer son instrumentation. L'ensemble des modules bas niveau est analogue à un alphabet qui va nous permettre d'écrire des mots : les sous-objectifs. Enfin, en enchaînant plusieurs sous-objectifs nous obtenons un objectif [39].

Il y a alors deux approches dans l'utilisation du vocabulaire : un approche ascendante qui va consister à construire les lettres et les mots qui vont refléter les capacités du véhicule et une approche descendante qui va consister à se servir de ces mots pour exprimer au système la mission qu'il devra effectuer.

Par exemple si l'on venait à ajouter sur un véhicule sous-marin un sonar à balayage, on construira alors le module bas niveau qui gérera ce capteur. On construit aussi un module qui sauvegarde les données produites par ce module. A partir de ces deux modules nous pouvons définir le sous-objectif "relevé topographique", cela signifie que lorsque l'ordonnanceur bas niveau reçoit ce sous-objectif, il va lancer périodiquement le module qui gère le capteur puis le module de sauvegarde.

En combinant ce sous-objectif et un autre sous-objectif qui permet de faire naviguer le véhicule en râteau sur une zone précise, nous pouvons créer l'objectif "bathymétrie".

Nous pouvons utiliser et réutiliser le vocabulaire disponible pour pouvoir construire les missions désirées par l'utilisateur.

Nous allons à travers un exemple mettre en évidence comment l'utilisation de ce vocabulaire va donner la possibilité au logiciel de contrôle de faire facilement de la gestion contextuelle de tâche (au sens robotique du terme).

Comme nous l'avons décrit précédemment, une mission est faite d'un ensemble d'objectifs que le système robotique doit atteindre. L'exécution de la mission revient à l'exécution de cette séquence logico-temporel d'objectifs.

Exemple de mission :

- Attente 30s
- Navigation jusqu'au point A

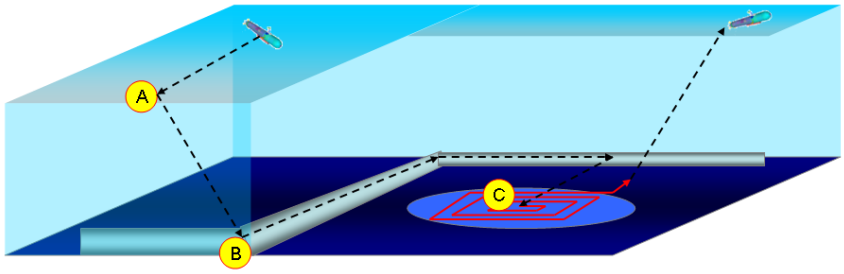


Figure 6.7 – Exemple de mission pour un véhicule sous-marin de type AUV

- Position GPS
- Inspection pipeline au point B + Relevé salinité
- Bathymétrie zone à partir du point C

La description de cette mission par l'utilisateur requiert la construction de cinq objectifs à savoir :

- Navigation,
- Position GPS,
- Inspection de pipeline,
- Relevé salinité,
- Bathymétrie.

La figure 6.7 représente la mission à accomplir par le véhicule sous-marin.
Les objectifs seront alors enchaînés par le superviseur global (figure 6.8).

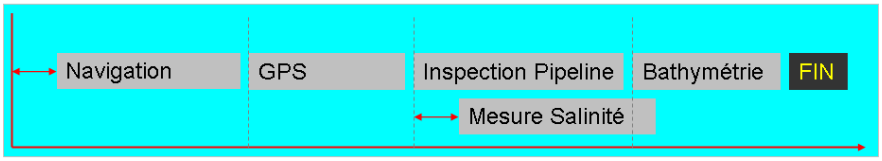


Figure 6.8 – Objectifs enchaînés par le superviseur global

Les informations concernant la manière dont le superviseur global devra agencer temporellement les objectifs pour accomplir la mission sont contenues dans les objectifs eux-mêmes. Ainsi chaque objectif renseigne sur l'événement qui doit l'engager et celui qui doit l'arrêter.

L'exécution d'un objectif peut être déclenchée après un temps fixé (potentiellement nul) compté suite à l'occurrence de deux types d'événements :

- Evénement traduisant un état particulier du système (exemple : véhicule à l'eau)
- Fin d'un autre objectif

D'autre part, l'arrêt d'un objectif se fera un temps fixé (potentiellement nul) après l'occurrence de :

- Evénement traduisant un état particulier du système

– Fin d'un autre objectif
– Fin propre

L'entité **objectif** peut alors être modélisée comme montré à la figure 6.9.

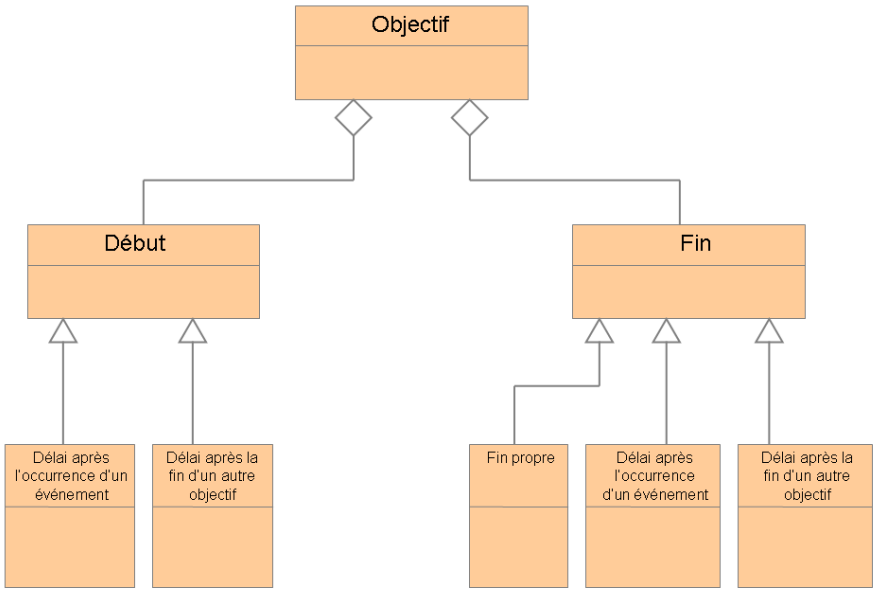


Figure 6.9 – Modélisation UML d'un objectif

Lors du déroulement de la mission, le superviseur global trouvera alors dans les objets **objectif** toutes les informations nécessaires à l'exécution et à l'enchaînement logico-temporelle des objectifs.

L'exemple précédent est un exemple idéal d'exécution de mission. La réalité nous impose d'envisager les cas d'échec d'objectifs durant une mission. Supposons que dans l'exemple précédent, le véhicule perde la trace du pipeline pendant le suivi. Une réaction possible serait de remonter en surface puis d'arrêter la mission en attendant l'intervention humaine (figure 6.10).

Toutefois, la perte d'un pipeline ne représente pas un danger direct pour un véhicule sous-marin. Elle peut seulement l'empêcher d'atteindre correctement un objectif. De plus ces situations sont assez courantes et prévisibles, dans le sens où l'opérateur peut savoir par exemple que les occultations de pipeline sont fréquentes dans la zone d'opération sans toutefois savoir à quels endroits exactement.

Intuitivement nous savons que l'on peut très bien poursuivre la mission en passant directement au dernier objectif : la bathymétrie (voir figure 6.11). Nous ne pouvons cependant dresser de règles générales dans le cas d'échec d'objectifs qui seraient appliquées par le superviseur global. En effet, l'échec d'un objectif dans certains cas peut invalider le suivant. L'opérateur ayant la capacité de déterminer le sens des objectifs composant la mission est plus à même de proposer une issue pour la mission en cas d'échec d'un objectif.

Le logiciel de contrôle doit pouvoir offrir à l'opérateur, lors de la programmation de

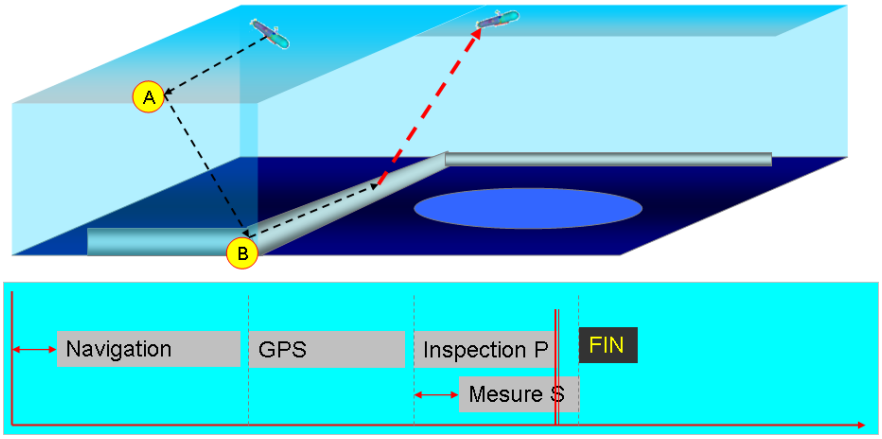


Figure 6.10 – Arrêt de la mission suite à l'échec d'un sous-objectif

la mission, la liberté de proposer une mission alternative à exécuter lorsque survient un problème donné.

Dans notre exemple, si en cas d'échec de l'objectif d'inspection de pipeline, nous voulons terminer la mission avec l'objectif bathymétrie, nous devrons construire un sixième objectif. Cet objectif sera un objectif bathymétrie, comme le cinquième, mis à part qu'il sera engagé seulement si l'objectif inspection pipeline est entré en échec. Les deux objectifs bathymétrie présents alors dans la mission diffèrent par leur condition de lancement.

Nous enrichissons alors le modèle d'un objectif en lui ajoutant un nouveau type de condition de lancement :

– Echec d'un autre objectif

et un nouveau type de condition d'arrêt :

– Echec.

6.3.2 De l'objectif au sous-objectif

Lorsqu'un objectif est sélectionné par le superviseur global pour être exécuté, il est décomposé en une séquence de sous-objectifs. Comme illustré à la figure 6.13(a) l'objectif **inspection de pipeline** est décomposé en la séquence de sous-objectifs suivante :

– plonger,
– aller à,
– rechercher pipeline,
– suivre pipeline.

Cependant, cette décomposition doit être adaptée au contexte. Dans l'exemple illustré à la figure 6.13(b) le véhicule se trouve déjà en profondeur. Pour accomplir l'objectif **inspection de pipeline** dans ce cas, la décomposition devient :

– aller à,
– rechercher pipeline,
– suivre pipeline.

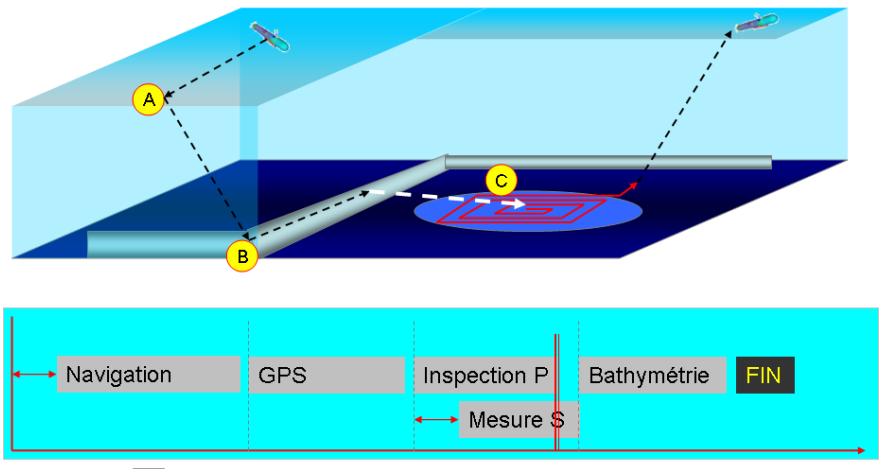


Figure 6.11 – Réaction à l'échec d'un objectif

Le sous-objectif **plonger** a été retiré. Nous sommes alors amené à rendre la décomposition d'objectif en sous-objectifs contexte-dépendante. Cette fonctionnalité est assurée par le superviseur local.

6.3.3 Les modules bas niveau

Comme nous l'avons expliqué précédemment, le bas niveau de l'architecture (voir figure 6.14) est en charge de la gestion de l'instrumentation de bord. Plus précisément, il doit exécuter les sous-objectifs en envoyant des commandes aux actionneurs et en évaluant l'état du système (acquisitions capteurs notamment).

Pour exécuter un sous-objectif donné, par exemple le sous-objectif **plonger**, plusieurs modules bas niveau devront être lancés successivement et périodiquement jusqu'à l'exécution complète du sous-objectif. Ces modules sont activés par des requêtes envoyées par l'ordonnanceur.

En consultant la base de données des sous-objectifs, l'ordonnanceur connaît, pour un sous-objectif donné l'ensemble des modules à activer, l'ordre d'activation, la durée d'exécution du module ainsi que la période à laquelle il devra répéter cette opération.

Ainsi pour exécuter le sous-objectif **plonger**, un module capteur gérant un sondeur sera activé pour pouvoir déterminer la distance au fond, un module action implémentant une loi de commande récupère cette valeur pour calculer une valeur de commande pour les actionneurs, enfin les modules actionneurs concernés appliquent la commande calculée.

Comme plusieurs sous-objectifs peuvent potentiellement être exécutés simultanément, plusieurs capteurs peuvent alors se trouver en fonctionnement en même temps. Cependant certains capteurs (la plupart du temps des capteurs acoustiques adjacents et/ou de fréquence proche ou multiple) ne peuvent effectuer leur mesure en même temps au risque de produire des interférences qui se répercuteront en erreurs de mesures sur les données acquises.

Lorsque des capteurs acoustiques ont des fréquences similaires, ils peuvent engendrer des interférences s'ils sont utilisés simultanément.

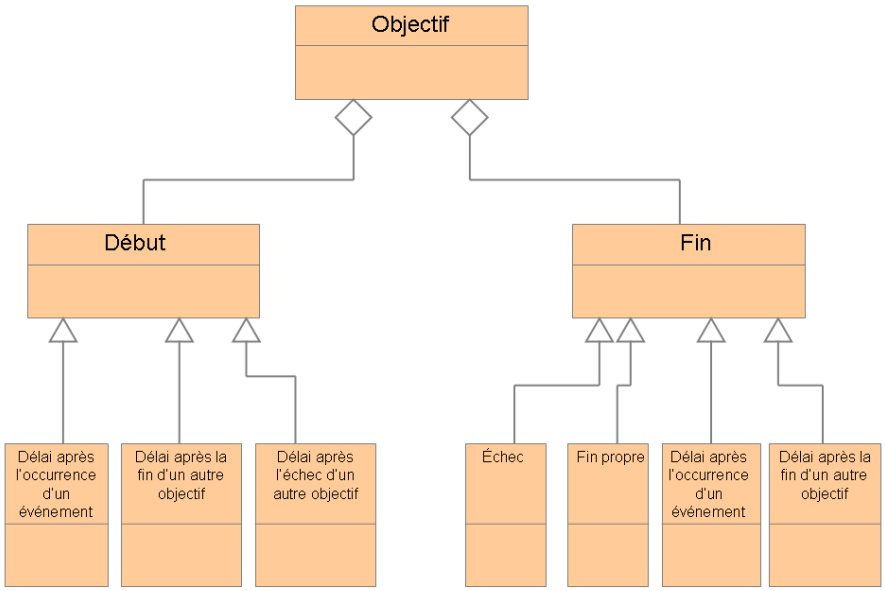


Figure 6.12 – Modélisation UML d'un objectif

Comme le montre la figure 6.15, les trois capteurs acoustiques présents à l'avant du véhicule, ne peuvent fonctionner ensemble car leurs champs d'insonification et d'écoute se recouvrent. Dans ce cas si deux sous-objectifs prêts à être exécutés ont besoin de deux capteurs parmi ceux-ci alors ces deux sous-objectifs ne peuvent pas être exécutés simultanément.

Cependant, nous pouvons constater que durant l'exécution d'un sous-objectif, les capteurs ne sont pas utilisés continûment mais de façon périodique. Ainsi une façon de permettre l'exécution de sous-objectifs requérants des capteurs interférents serait d'activer ces capteurs (donc les modules capteurs bas niveau qui les gèrent) séquentiellement.

On ne peut prévoir à l'avance les ensembles de sous-objectifs qui vont être amenés à être exécutés simultanément durant la mission. Ainsi la prise en compte des contraintes d'interférences capteurs dans l'activation des modules bas niveau sera effectuée en temps réel par l'ordonnanceur.

6.4 Approche synchrone/asynchrone

Nous avons vu que les niveaux supérieurs (superviseur global et superviseurs locaux) gèrent le lancement et l'arrêt des objectifs et des sous-objectifs. Le lancement et l'arrêt de ces derniers se fait selon certains événements soit décrits par l'utilisateur soit imposés (ex : entrée d'eau, énergie faible,...) et surveillés par les superviseurs concernés.

Lorsque le superviseur global reçoit une mission, il consulte une base de données (figure 6.6) qui contient la liste des objectifs à lancer pour exécuter la mission donnée. Cette liste se compose des conditions de lancement de chaque objectif ainsi que de ses conditions d'arrêt.

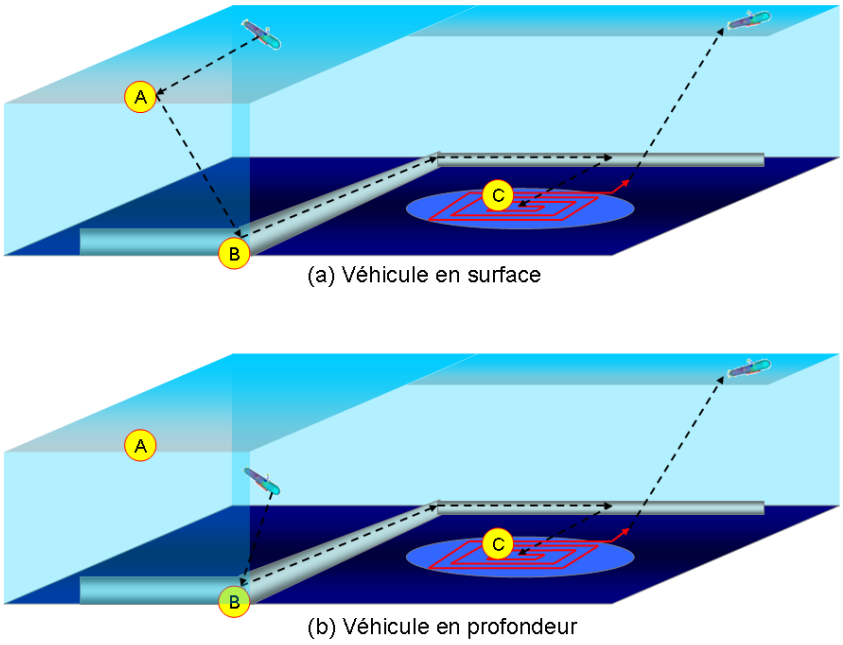


Figure 6.13 – Décomposition contextuelle des objectifs

Format général d'une mission :		
[condition de lancement]	nom objectif 1	[condition d'arrêt]
[condition de lancement]	nom objectif 2	[condition d'arrêt]
	...	
[condition de lancement]	nom objectif n	[condition d'arrêt]

De manière analogue, les superviseurs locaux consultent une base de données qui contient la décomposition des objectifs en sous-objectifs accompagnés de leurs conditions de lancement et d'arrêt.

Une condition de lancement (resp. d'arrêt) pour un objectif ou pour un sous-objectif représente les conditions à satisfaire pour déclencher (resp. arrêter) l'objectif ou le sous-objectif concerné. Ces conditions concernent l'état interne ou externe (environnement) du système robotique commandé ainsi que le temps. L'état du véhicule est traduit par les capteurs (proprioceptifs et extéroceptifs) présents à bord. Les modules superviseurs ont alors besoin de pouvoir analyser les données de ces capteurs afin de pouvoir évaluer ces équations logiques.

Cependant ces modules n'ont pas directement accès à l'instrumentation de bord, d'autres modules sont dédiés à cela et l'utilisation de ces capteurs (dans le cas où ils seraient interférents) est soumise à des règles que seul l'ordonnanceur bas niveau est en mesure d'appliquer. D'autre part, les états du système à surveiller sont discrets, ils peuvent être détectés dans un flux continu de données. Par exemple si l'on veut détecter l'état : "obstacle en vue", nous n'avons pas besoin de toutes les valeurs du capteur de distance mais seulement d'une information ponctuelle qui indiquera le passage de cette valeur en dessous/dessus d'un certain seuil. On appellera ce type d'information "un événement".

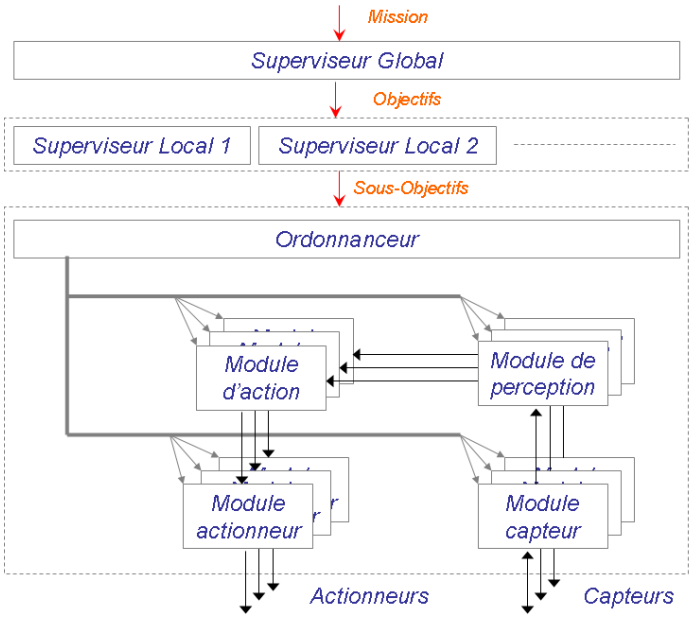


Figure 6.14 – Architecture logicielle de commande

Ainsi l'analyse des événements produits par les modules bas niveau de l'architecture permet de connaître l'état du véhicule. Cela suppose d'une part que le superviseur concerné s'abonne à l'événement auprès du module bas niveau qui le produit et d'autre part que ce module soit en activité.

En plus des événements d'état générés par les modules bas niveau, les équations logiques vont aussi comporter des événements de temps. Pour une mission, on pourra indiquer dans les conditions de lancement des objectifs qui la composent, des événements de temps avec comme référence le début de la mission. Par contre les conditions d'arrêt peuvent aussi contenir des événements de temps, mais avec comme référence le début de l'objectif.

Nous avons vu au chapitre 5 que les événements étaient produits par des modules à des ports spécifiques, si bien qu'un nom de module accompagné d'un numéro de port peut identifier un événement précis au sein de l'architecture. La définition d'une mission par exemple sera de la forme suivante :

Mission :		
[nomModule :N°port $\otimes$ nomModule :N°port $\oplus$ temps=α s]	nom objectif 1	[duree=η s $\oplus$ nomModule2 :N°port x date=γ s]
[nomModule :N°port $\oplus$ temps=δ s]	nom objectif 2	[temps=τ s $\oplus$ nomModule2 :N°port]
...		

Les opérateurs "$\oplus$" et "$\otimes$" utilisés dans les équations logiques vont traduire respectivement l'alternative et la simultanéité de deux événements. En ce qui concerne l'alternative, l'équation 6.1 signifie que l'équation donnera un résultat vrai si l'un des deux événements vient de se produire. Par contre pour la relation "simultanéité" (équation 6.2), donnera un résultat vrai lorsque les deux événements evt1 et evt2 se produisent, qui elle n'est jamais vérifiable (la simultanéité n'existant pas en événementiel).

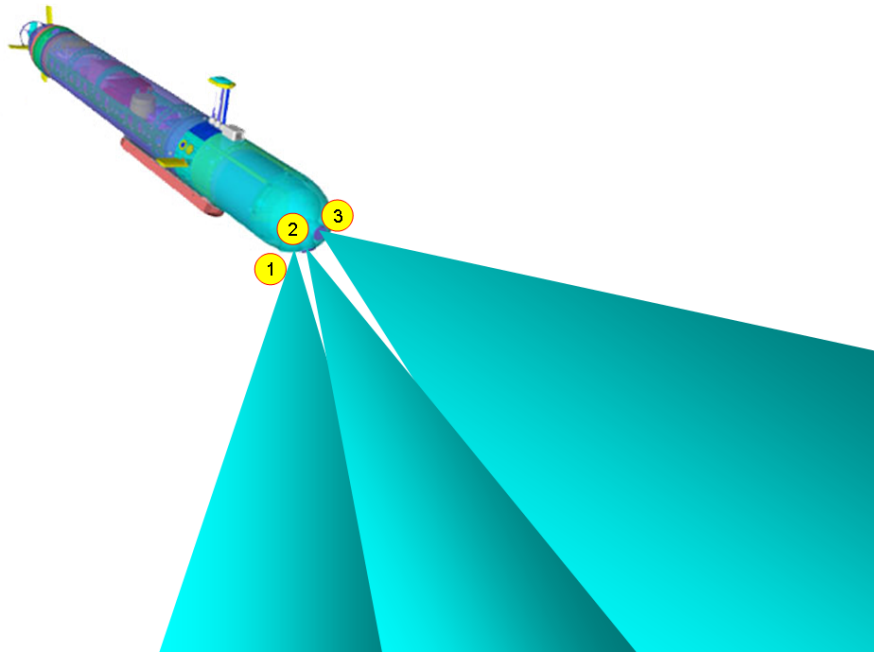


Figure 6.15 – Interférence entre les capteurs acoustiques

$$evt1 \oplus evt2 \tag{6.1}$$

$$evt1 \otimes evt2 \tag{6.2}$$

En effet, les modules du niveau supérieur de l'architecture fonctionnent de façon événementielle, c'est à dire que les traitements sont engagés dès qu'un événement est reçu (nouvelle mission, nouvel objectif, événement provenant d'un module ou événement de temps). Donc l'équation logique sera évaluée dès la réception du premier événement (evt1) puis réévaluée à la réception du second. A la réception de l'un, l'autre n'est pas encore reçu (ou du moins pas encore traité) ce qui fait que les équations comportant un "et" logique seront toujours fausses.

Il y a des cas, pourtant où l'on voudrait exprimer une simultanéité entre deux événements. Si nous prenons l'exemple d'un véhicule sous-marin qui doit faire des relevés topographiques de la surface d'une source d'eau douce sous-marine. Les relevés devant être faits au delà d'une certaine profondeur. Nous voudrions alors que l'objectif **relevé topographique** soit lancé une fois que la profondeur dépasse une valeur donnée et que la salinité (détection de la zone d'eau douce) descend en deçà d'une certaine valeur. Les événements "seuil de profondeur atteint" et "seuil de salinité atteint" ne seront pas reçus simultanément.

Pour traduire l'idée intuitive qu'on se fait de la simultanéité des événements, on doit pouvoir faire persister les événements sur lesquels on veut raisonner. On devra alors préciser sur les événements liés par un "et" logique, la condition de fin de persistance. Cette dernière pouvant être une durée ou un autre événement.

Dans l'équation 6.3, l'événement "evt1" sera mémorisé 5s après sa réception. Si l'"evt2" est reçu dans les 5 secondes après la réception de l'événement "evt1", l'équation 6.3 sera

vraie. Dans l'équation 6.4 la fin de persistance de l'événement "evt2" est signalée par l'événement "evt3".

$$evt1(5s) \otimes evt2 \tag{6.3}$$

$$evt1 \otimes evt2(evt3) \tag{6.4}$$

6.5 Langage de programmation de l'architecture basé sur un vocabulaire

L'architecture que nous proposons est hiérarchisée en deux niveaux, le niveau décisionnel contenant le Superviseur Global et les Superviseurs Locaux et le niveau Exécutif. Chacun de ces niveaux gère les décisions relatives à son niveau de compétence. Le Superviseur Global agira sur le cours de la mission sur un plan très agrégé comme l'arrêt d'une phase de la mission ou le basculement vers une procédure d'urgence par exemple. Le Superviseur Local, selon la ressource matérielle à laquelle il est dédié, agira de façon à accomplir les décisions émanant du Superviseur Global en transmettant au niveau exécutif les différents traitements qu'il doit mettre en oeuvre. Le niveau Exécutif se charge de réaliser les traitements voulus par le niveau supérieur (Superviseur Locaux) tout en respectant les possibilités de l'engin et les différentes contraintes matérielles.

Les décisions prises à chaque niveau de l'architecture sont fonction des capacités de l'engin et de l'intention de l'opérateur. Chaque système robotique possède des capacités qui lui sont propres et les buts à atteindre vont varier d'une mission à l'autre. Ce qui nous amène à dire que si les capacités de l'engin et l'ensemble des buts qu'il peut atteindre faisaient partie intégrante de l'architecture alors celle-ci (dans sa globalité) sera limitée en terme d'évolutivité et de réutilisabilité. En effet, l'augmentation ou la réduction des capacités de l'engin (par l'ajout ou la suppression d'instruments embarqués par exemple) ou l'orientation vers de nouvelles applications du robot ne doit pas passer par une modification de l'architecture.

Dans notre approche nous avons établi une séparation entre le cadre de raisonnement et de prise de décision d'une part et l'ensemble des connaissances liées au système physique et ses applications possibles d'autre part. Cela se traduit dans un premier temps par la réification de la décision au sein de l'architecture, qui nous amène à l'établissement d'un vocabulaire basé sur un "alphabet" traduisant l'ensemble des capacités du robot d'une part, et d'un ensemble de "mots" mis à disposition de l'opérateur pour constituer le "texte" traduisant son intention d'autre part. Dans un deuxième temps, la définition de ce vocabulaire a été séparée de l'architecture comme illustré à la figure 6.6. Dans cette figure les différents niveaux de prise de décision consultent leurs bases de données respectives pour obtenir les règles d'interprétation des intentions qu'ils reçoivent.

Ce vocabulaire a pour alphabet les **module**s du niveau exécutif, pour mots les **sous-objectif**s, pour phrases les **objectif**s et pour texte la **mission** de l'opérateur. Une mission est constituée d'un ensemble d'objectifs avec des règles d'enchaînement, un objectif est constitué d'un ensemble de sous-objectifs avec également le même type de règle et enfin un sous-objectif est constitué d'une suite de modules à exécuter dans un ordre précis.

Les bases de données de vocabulaire sont l'élément qui va identifier (personnaliser) l'architecture pour un système robotique donné. Tout changement au niveau des capacités

robotique se répercutera dans l'adaptation de ce vocabulaire. Pour chaque niveau, nous définissons des règles "grammaticales" pour la construction du vocabulaire :

Définition d'un module :
- nom donné au module,
- paramètres temporel du module ($< C^-, L, C^+ >$ ou $< C >$ cf. chapitre 7),
- listes des arguments du module.

Définition d'un sous-objectif à partir de modules :
- nom donné au sous-objectif,
- listes des arguments du sous-objectif,
- noms des modules mis en oeuvre,
- listes des paramètres à passer au module avant son exécution,
- ordre d'enchaînement de l'exécution des modules (cf. chapitre 7),
- flux de données intermodule à établir (cf. 5),
- période de répétition (cf. chapitre 7),
- délai critique (cf. chapitre 7).

Définition d'un objectif à partir de sous-objectifs :
- nom donné à l'objectif,
- listes des arguments de l'objectif,
- noms des sous-objectifs mis en oeuvre,
- équation logique de lancement de chaque sous-objectif (cf. section 6.4),
- équation logique d'arrêt de chaque sous-objectif (cf. section 6.4),
- liste des paramètres à passer à chaque sous-objectif avant son exécution.

Définition d'une mission à partir d'objectifs :
- nom donné à la mission,
- listes des arguments de la mission,
- noms des objectifs mis en oeuvre,
- équation logique de lancement de chaque objectif (cf. section 6.4),
- équation logique d'arrêt de chaque objectif (cf. section 6.4),
- liste des paramètres à passer à chaque objectif avant son exécution.

Si nous prenons l'exemple d'un robot sous-marin auquel nous ajouterions une caméra vidéo pour filmer les épaves sous-marines, dans un premier temps nous devons créer (ou récupérer) un module que nous appellerons **video** gérant cette caméra et un module de sauvegarde de données. Ce module caméra accepte un paramètre (un entier) de sélection de mode photo (0)/mode vidéo (1) et un paramètre (un entier) de sélection du taux de rafraîchissement de l'image. Nous aurons alors les définitions suivantes dans les bases de données correspondantes :

Définition des modules :
video(10ms){entier resolution, entier mode} ; sauvegarde(5ms){} ; etc...

Les 10 ms indiquent la durée maximale d'exécution estimée du module **video** (cf. chapitre 7). La déclaration du module est suivie de la liste des paramètres qu'il accepte dans l'ordre. L'ordre des paramètres indique que le paramètre de **resolution** devra être envoyé au port de paramétrage n°0 du module et le paramètre **tauxRafraichissement** au port n°1 (cf. chapitre 7).

Nous pouvons maintenant définir un sous-objectif **acquisition_video** qui va se baser sur les modules précédents :

Définition du sous-objectif acquisition video :

```
acquisition_video(entier resolution)
{
        video(resolution, 1);
        sauvegarde();
        video -> sauvegarde;
        video :0 -> sauvegarde :0;
        PERIODE = 40 ms;
        DELAI_CRITIQUE = 20 ms;
}
```

L'objectif **acquisition_video** aura un argument **resolution** qui sera passé en paramètre au module vidéo par contre la valeur du paramètre mode est inscrite directement dans la définition du sous-objectif étant donné que pour exécuter le sous-objectif d'acquisition vidéo il est nécessaire que le module video soit configuré en mode vidéo.

Pour définir un objectif **inspection_epave** nous devons réutiliser d'autres sous-objectifs qui auraient été développés pour la navigation propre du véhicule :

Définitions de l'objectif inspection_ epave :

```
inspection_epave(entier lattitude, entier longitude, entier altitude, entier resolution)
{
        1 : []              plonger()                          [profondeur :2];
        2 : [fin(1)]        naviguer(lattitude, longitude, altitude)   [commande :1];
        3 : [fin(2)]        inspection_epave(resolution)       [];
        4 : [fin(2)]        rateau(altitude)                   [];
}
```

Nous reprenons ici la syntaxe de déclaration des équations logique d'événements définie à la section 6.4. Notre objectif ainsi défini donne la possibilité à l'utilisateur de spécifier une zone géographique où se trouve une épave et de choisir la résolution de l'image vidéo.

La déclaration ci-dessus signifie que pour accomplir l'objectif **inspection_epave** :

- Nous devons lancer le sous-objectif **plonger** puis nous abonner à l'événement **torpille sous l'eau** produit au port n°2 du module **profondeur**.
- Le sous-objectif **naviguer** sera lancé lorsque le sous-objectif **plonger** (de la ligne taguée "1 :") se sera terminé et il s'arrêtera dès que le module **commande** émettra un événement "position atteinte" sur son port 1.
- Les sous-objectifs **inspection_epave** et **rateau** seront lancés tous deux dès que le sous-objectif navigation (de la ligne taguée "2 :") s'est arrêté et ils prolongeront leur exécution jusqu'à la fin de l'objectif.

Enfin nous pouvons introduire le nouvel objectif ainsi obtenu dans une mission que nous appellerons **mission_lirmm** :

Définitions de la mission mission_lirmm :

```
mission_lirmm(entier latitude, entier longitude, entier altitude, entier resolution)
{
      1 : []              inspection_pipeline(98, 12, 110)          [duree=1h] ;
      2 : [fin(1)]          inspection_epave(128, 236, 100)         [duree=1h] ;
      3 : [fin(2)]           surface(latitude, longitude)           [profondeur :1] ;
}
```

Les bases de données qui vont contenir toutes les définitions des éléments de vocabulaire de l'architecture sont amenées à être enrichies au gré des évolutions du matériel embarqué ou de l'apparition de nouvelles applications pour le système robotique. Un des points essentiel est de permettre au maximum la réutilisation en offrant la possibilité de :

- réutiliser les modules existants pour former un nouveau sous-objectif,
- réutiliser les sous-objectifs existants pour former un nouvel objectif,
- réutiliser les objectifs existants pour former une nouvelle mission,

Résumé du chapitre 6 :

- Nous proposons une architecture logicielle basée sur une composition (particulière) des modules définis au chapitre 5.
- Notre architecture est hiérarchisée en deux niveaux : un niveau décisionnel comprenant un Superviseur Global (gére les étapes de la mission) et un Superviseur Local (gére les sous-étapes de la mission) et un niveau exécutif (contient un ordonnanceur qui permet de determiner les dates d'exécution de chaque module bas niveau et ainsi de répondre à différentes contraintes).
- Chaque niveau prend les décisions relatives à son niveau d'abstraction. Chaque niveau applique les décisions de son niveau supérieur en fonction du contexte.
- On définit un vocabulaire qui nous permet d'exprimer les capacités du système robotique (nom de modules bas niveau), les intentions de l'opérateur (objectifs) et la manière dont on doit employer les capacités de l'engin pour accomplir les intentions de l'opérateur (sous-objectif).

Chapitre 7

Le niveau exécutif

7.1 Introduction

Notre architecture logicielle de contrôle est constituée de deux niveaux : un niveau décisionnel comprenant le superviseur global et les superviseurs locaux et un niveau exécutif comprenant un ordonnanceur et les modules bas niveau. Les superviseurs prennent des décisions concernant la mission qu'ils transmettent au niveau bas de l'architecture. Le bas niveau de l'architecture est en charge de rendre effectif les décisions haut-niveau (commande des actionneurs) et de rendre compte de l'état extéroceptif et proprioceptif du système robotique.

Le bas niveau constitue une abstraction de la structure matérielle embarquée à bord du véhicule. En effet, comme nous l'avons expliqué au chapitre précédent, nous avons donné existence aux objets "sous-objectif" qui traduisent les intentions combinées des niveaux supérieurs à destination du bas niveau. Le bas niveau est en charge de prendre les mesures nécessaires et de répondre aux contraintes imposées par le matériel afin d'exécuter ces sous-objectifs ou à défaut d'informer les niveaux supérieurs en cas d'impossibilité.

Les contraintes gérées au niveau inférieur sont : partage de l'unité de calcul entre les différents traitements, exclusion mutuelle dans l'utilisation de l'instrumentation de bord, respect des périodes d'échantillonnage. Les dates d'activation des modules bas niveau doivent être déterminées avec précision pour respecter les différentes contraintes.

Nous allons exposer plus en détails dans ce chapitre les contraintes imposées au bas niveau et proposer une méthode, basée sur un algorithme d'ordonnancement des modules que nous appelerons indifférement tâches. Enfin nous expliquerons la place que devra occuper l'ordonnanceur dans l'architecture.

7.2 Les contraintes bas niveau

7.2.1 Les contraintes temporelles

Le bas niveau de l'architecture est composé de modules gérant l'instrumentation (capteurs et actionneurs) et de modules effectuant des traitements informatiques sur les don-

nées produites par ces derniers. Les modules bas niveau doivent être activés de manière périodique pour pouvoir maintenir le système dans l'état voulu ou le faire évoluer vers un autre état.

L'activité bas niveau devra traduire les décisions du haut-niveau de l'architecture. Ces décisions sont reçues sous la forme de sous-objectifs. Un sous-objectif traduit une activité précise à mener par le système qui va requérir l'activation périodique d'un ensemble de modules bas niveau pendant un certain temps. Un module bas niveau particulier qu'on appelle "ordonnanceur" est en charge d'analyser les sous-objectifs transmis par le niveau supérieur et d'activer les modules nécessaires à leur réalisation.

Si nous prenons l'exemple du sous-objectif **plonger**, son exécution devra donner lieu à l'activation périodique de 3 modules : le module gérant le capteur de pression (pour calculer la profondeur), le module loi de commande et le module gérant les actionneurs. Cette suite d'activations de modules, depuis la perception jusqu'à l'actionnement, doit respecter certaines règles imposées par les lois de l'automatique. En effet, pour assurer la stabilité de la commande d'un véhicule, il faut pouvoir délivrer les commandes aux actionneurs à une périodicité précise. Il faut en outre que ces commandes soient calculées à partir de données capteurs récentes pour ne pas introduire un retard entre ce qui est perçu par le système et la "réaction" qu'il produit. Ces contraintes seront appelées les contraintes de dates d'exécution.

Les différents sous-objectifs que peut exécuter le système piloté, sont le résultat de l'exécution d'une suite ordonnée de modules (voir chapitre 4). Pour pouvoir exécuter le sous-objectif désiré, on doit enchaîner l'exécution des modules correspondants dans un ordre précis. La description d'un sous-objectif dans la base de données consultée par l'ordonnanceur doit aussi préciser les relations d'ordre entre les modules. Ces contraintes seront appelées les contraintes de précédence.

7.2.2 Les contraintes matérielles

Comme nous l'avons expliqué au chapitre précédent, certains capteurs (la plupart du temps des capteurs acoustiques adjacents), ne peuvent effectuer leurs mesures en même temps au risque de produire des interférences qui se répercuteront en erreurs de mesure sur les données acquises. Il convient alors d'employer l'instrumentation interférente à des moments différents.

Certains capteurs permettent un fonctionnement sur requête, donc on peut tout à fait contrôler leur date de fonctionnement. S'il s'agit par exemple de capteurs acoustiques, on peut les commander de façon à ce qu'ils insonifient à des moments différents. Pour ce faire on activera les modules gérant ce type d'instrumentation de façon séquentielle.

Par contre si les capteurs ont un fonctionnement continu, s'ils effectuent de leur propre chef des mesures depuis leur allumage jusqu'à leur extinction, on ne pourra pas trouver de méthode logicielle permettant de gérer les cas de conflit d'utilisation simultanée de ce type d'instrumentation.

Enfin, il est à noter que la perception basée sur une instrumentation acoustique peut être décomposée en trois temps :

1. activation du capteur (besoin du CPU)
2. propagation des ondes dans le milieu (pas besoin du CPU)
3. collecte et traitement des données (besoin du CPU)

7.3 Formalisation du problème d'ordonnancement

Comme nous l'avons décrit au chapitre 4, lorsqu'un module est activé, il effectue son traitement puis attend une nouvelle activation. Aucun module ne boucle sur lui-même. L'exécution d'un module sera modélisée par une tâche (au sens processus) avec une durée d'exécution.

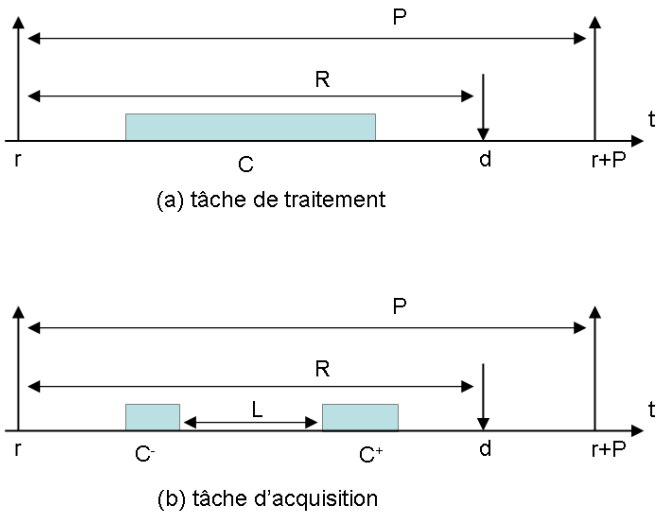


Figure 7.1 – Paramètres temporels des tâches de traitement et des tâches d'acquisition

Nous distinguons l'exécution d'un module contenant un simple traitement informatique, qu'on appellera une **tâche de traitement** et l'exécution d'un module gérant un moyen de perception, qu'on appellera des **tâches d'acquisition**[40]. Comme illustré à la figure 7.1.a, une tâche de traitement ressemble à une tâche informatique définie par quatre paramètres temporels :

- r : date de réveil
- C : durée totale
- R : délai critique
- P : période

Une telle tâche sera notée $T :< r, C, R, P >$ et peut être préemptée.

Une tâche d'acquisition est divisée en trois parties (voire figure 7.1.b) et définie par six paramètres temporels dont :

- C^- : durée d'envoi de la requête d'insonification au capteur
- C^+ : durée de récupération des données capteur
- L : latence de réponse du capteur

Pendant la durée "L" d'une tâche d'acquisition, une autre tâche peut être exécutée (le processeur étant libre) mais doit s'arrêter avant l'occurrence de la dernière partie de cette tâche noté "C^+". Une telle tâche sera notée $A :< r, C^-, L, C^+, R, P >$ et ne pourra être ni préemptée ni retardée puisqu'elle doit se synchroniser avec l'instrumentation.

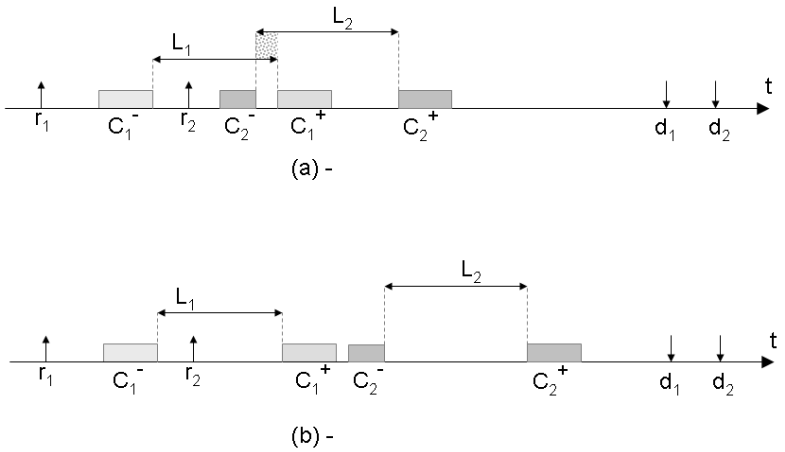


Figure 7.2 – Exécution de tâches exclusives

Lorsque deux modules, ne peuvent s'exécuter simultanément car ils gèrent deux instrumentations interférentes, on dit que les tâches qui résultent de leurs exécutions sont mutuellement exclusives. Pour éviter les conflits (voir figure 7.2.a), l'une ne pourra être lancée que si l'autre a terminé son exécution (figure 7.2b).

L'exécution d'un sous-objectif va consister en l'exécution d'une suite de tâches de façon périodique jusqu'à la fin du sous-objectif. Cette suite de tâche commencera son exécution à la réception **r** du sous-objectif par l'ordonnanceur et devra se terminer avant la date **r**+**R**. Cette séquence sera répétée avec une période **P**. Cette séquence répétée qui représente l'exécution d'un sous-objectif sera appelée une **sous-configuration**.

Une sous-configuration sera caractérisée par trois paramètres temporels r, R et P et un ensemble de tâches d'acquisition et de traitements :

$$\text{Sc} :<\text{r, R, P, E}> \text{ avec E}=\{A_1, A_2, ...A_p, T_1, T_2, ...T_q\}$$

Certains modules doivent s'exécuter dans un ordre précis ce qui implique que l'ensemble des tâches **E** est un ensemble partiellement ordonné par une relation binaire $\prec$:

$\forall(M, N) \in E^2$, si $M \prec N$ alors l'exécution de M doit être terminée avant que l'exécution de N puisse commencer.

Nous modéliserons l'ordre partiel d'un ensemble de tâches E par un graphe orienté G dont les sommets représentent les tâches A_i et T_i et les arcs <M,N> si et seulement si $M \prec N$ [41]. La figure 7.3, représente la modélisation de l'ordre partiel dans la sous-configuration suivante :

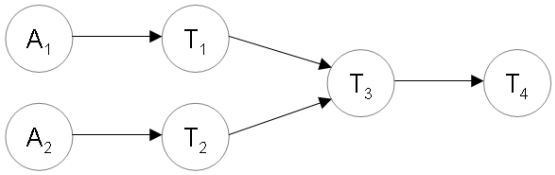


Figure 7.3 – Graphe de précédence d'un ensemble de tâches d'une sous-configuration

$$\begin{aligned} Sc &: < r, R, P, E > \\ \text{avec } E &= \{A_1, A_2, T_1, T_2, T_3, T_4\} \\ \text{et les relations d'ordres} &: A_1 \prec T_1 \\ & \quad T_1 \prec T_3 \\ & \quad A_2 \prec T_2 \\ & \quad T_2 \prec T_3 \\ & \quad T_3 \prec T_4 \end{aligned}$$

Considérons l'ensemble des sous-objectifs $E_{So} = So_1, So_2, ..., So_n$ à exécuter à une moment donné et l'ensemble des sous-configurations correspondantes $E_{Sc} = Sc_1, Sc_2, ..., Sc_n$ avec :

$$Sc_1 :< r_1, R_1, P_1, E_1 > Sc_2 :< r_2, R_2, P_2, E_2 > ... Sc_n :< r_n, R_n, P_n, E_n >$$

Exécuter l'ensemble E_{So} des sous-objectifs revient à trouver un ordonnancement valide pour l'ensemble de tâches $S = E_1 \cup E_2 \cup ... \cup E_n$ obéissant aux conditions suivantes :

1. $\forall i, \forall (M, N) \in E_i^2 si M \prec N$, alors M doit se terminer avant que N commence
2. $\forall (M, N) \in E_i^2$ si M et N sont mutuellement exclusives, alors M et N ne peuvent être lancées en même temps.

7.4 Solution apportée au problème d'ordonnancement

A chaque instant, l'ordonnanceur est en présence d'un ensemble de tâches de traitement et d'acquisition à exécuter en tenant compte des différents paramètres temporels et des contraintes de précédence et d'exclusivité [42].

7.4.1 Tâches de traitement

L'ordonnancement des tâches de traitement entre dans la catégorie des ordonnancements monoprocesseur de tâches périodiques préemptibles avec contrainte de précédence. Cet ordonnancement est réalisable en temps réel et de façon optimale par plusieurs algorithmes dont **Earliest Deadline** et **Least Laxity** [35].

L'algorithme Earliest Deadline, attribue la plus grande priorité à la tâche qui a l'échéance (d) la plus proche. L'algorithme Least laxity donne la plus grande priorité la tâche dont la laxité est la plus petite. La laxité est le temps restant avant l'échéance de la tâche.

Nous avons choisi l'algorithme Earliest Deadline parce qu'il donne lieu à moins de commutations de contextes que l'algorithme Least Laxity (comparaison possible entre les figures 7.4 et 7.5, basées sur le même exemple).

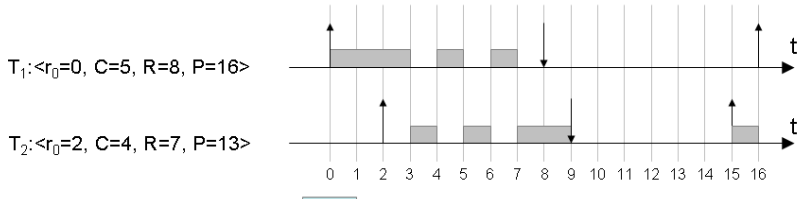


Figure 7.4 – Ordonnancement Least Laxity pour des tâches de traitement

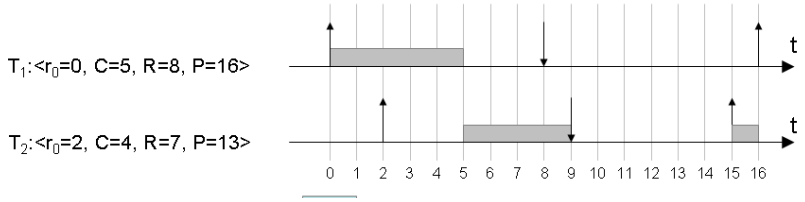


Figure 7.5 – Ordonnancement Earliest Deadline pour des tâches de traitement

7.4.2 Tâches d'acquisition

Les tâches d'acquisition possèdent des caractéristiques temporelles particulières qui n'ont pas été traitées dans la littérature : une thèse a été lancée sur ce sujet au laboratoire [43] et nous ne formulerons qu'une solution non optimale. De plus la contrainte d'exclusion mutuelle introduit une complexité supplémentaire dans la résolution des ordonnancements.

Pour économiser du temps processeur, il est judicieux de lancer simultanément l'exécution des tâches d'acquisition qui n'ont pas de contraintes d'exclusion entre elles. Considérons l'ensemble des tâches d'acquisition suivantes :

- $A_1 =< r = 0, C^- = 1, C^+ = 13, ... >$
- $A_2 =< r = 0, C^- = 1, C^+ = 7, ... >$
- $A_3 =< r = 0, C^- = 1, C^+ = 7, ... >$
- $A_4 =< r = 0, C^- = 1, C^+ = 17, ... >$
- $A_5 =< r = 0, C^- = 2, C^+ = 8, ... >$

Ces tâches présentent des exclusions mutuelles qui sont représentées par le graphe de compatibilité de la figure 7.6. La présence d'une arête entre deux sommets indique que les tâches concernées ne sont pas en exclusion. Nous pouvons constater à la figure 7.7.b qu'un meilleur ordonnancement est trouvé en s'attachant à exécuter en même temps les tâches compatibles.

Les tâches compatibles sont celles qui sont reliées par des arêtes sur le graphe (figure 7.6). Pour qu'un groupe de tâches puisse être exécuté simultanément, il faut que toutes les tâches qui le constituent soient reliées une à une dans le graphe de compatibilité. On dit alors que ces tâches forment une clique dans le graphe de compatibilité.

Evidemment plus le nombre de tâches reliées est grand plus le nombre de tâches qui vont être lancées en parallèle est grand et donc plus l'ordonnancement donne de bons

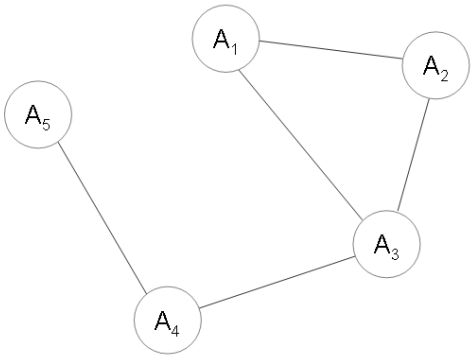


Figure 7.6 – Graphe de compatibilité des tâches (ou modules)

résultats. La recherche de l'ordonnancement des tâches d'acquisition passe alors par la recherche des plus grandes cliques dans le graphe de compatibilité.

Cependant la recherche de cliques dans le graphe de compatibilité est un problème NP-complet. Une étude est en cours pour déterminer s'il peut exister un algorithme optimal pour ce problème d'ordonnancement voir [43].

Pour le problème d'interférence qui se pose en robotique, nous chercherons à mettre en place un algorithme qui peut nous donner une solution faisable en un temps borné (utilisation en temps réel) même s'il n'est pas optimal (non fondé sur la recherche de cliques).

Comme les tâches d'acquisition ne peuvent pas être préemptées, le seul cas où deux de ces tâches entreront en compétition est lorsqu'elles doivent commencer en même temps, c'est à dire que leurs dates de réveil sont identiques. Dans ce cas nous élirons la tâche qui a l'échéance la plus proche. On utilisera alors l'algorithme Earliest Deadline pour ordonnancer les tâches d'acquisition, ce qui n'est pas une stratégie optimale (voir figure 7.8).

7.4.3 Tâches de traitement et tâches d'acquisition

Finalement entre une tâche d'acquisition et une tâche de traitement la priorité est calculée en prenant en compte deux paramètres à savoir le type de tâche (traitement ou acquisition) et l'échéance de la tâche.

Nous proposons la formulation suivante :

$$p_i = w_1 * d_i + w2 * \lambda_i \tag{7.1}$$

L'équation 7.1 donne la priorité relative d'une tâche T_i :
- p_i est la priorité qu'on donnera à la tâche,
- w_1 est le poids de l'échéance,
- d_i est l'échéance de la tâche,
- w_2 est le poids du type de tâche,
- λ_i est le type de la tâche ($\lambda_i = 0$ si la tâche T_i est une tâche de traitement, sinon $\lambda_i = 1$)

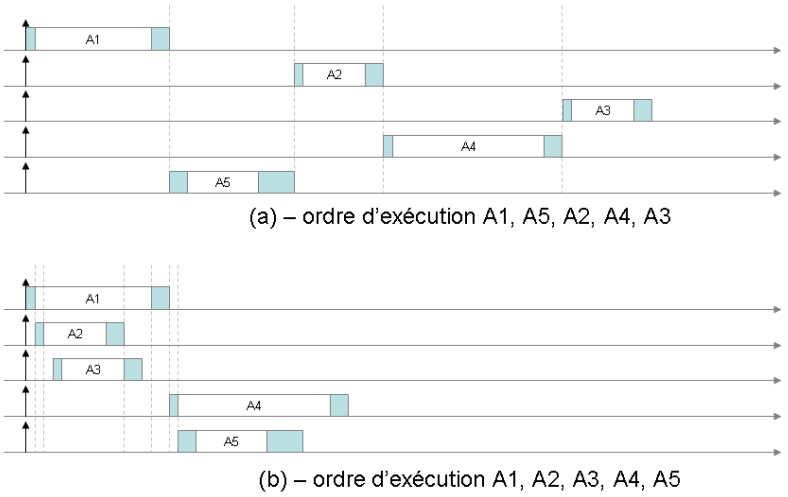


Figure 7.7 – Ordonnancement des tâches d'acquisition

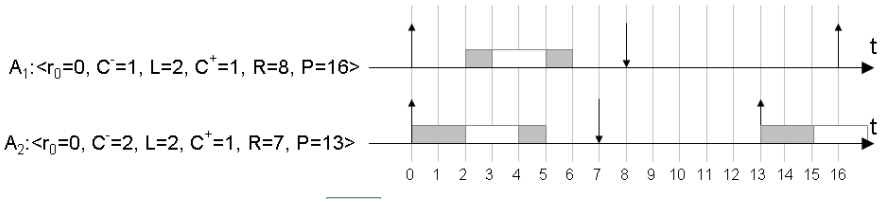


Figure 7.8 – Ordonnancement Earliest Deadline pour les tâches d'acquisition

Les meilleurs résultats ont été obtenus lorsque w_1 vaut -1 et lorsque w_2 est égal à la moyenne des durées des tâches de traitement (voir figures 7.9 et 7.10).

Lorsqu'une tâche de traitement et une tâche d'acquisition sont réveillées simultanément, les meilleurs résultats sont obtenus en privilégiant les tâches d'acquisition (figure 7.10). Cela est dû au fait que comme les tâches d'acquisition contiennent une zone notée L durant laquelle le processeur est rendu disponible, les tâches de traitement peuvent s'y exécuter (même partiellement).

7.5 Position de l'ordonnanceur au sein de l'architecture

Le module ordonnanceur est une entité architecturale qui va permettre d'articuler les niveaux purement décisionnels avec le niveau exécutif. Placé juste en dessous des super-

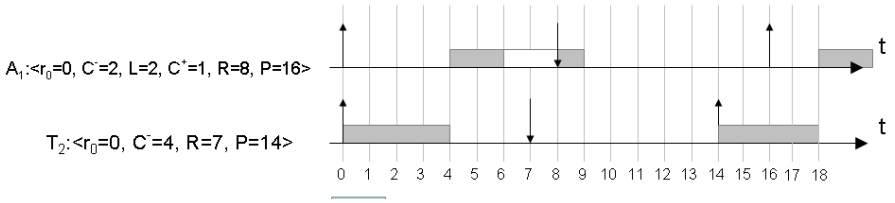


Figure 7.9 – Ordonnancement de tâches de traitement et d'acquisition avec $w_1 = -1$ et $w_2 = 0$

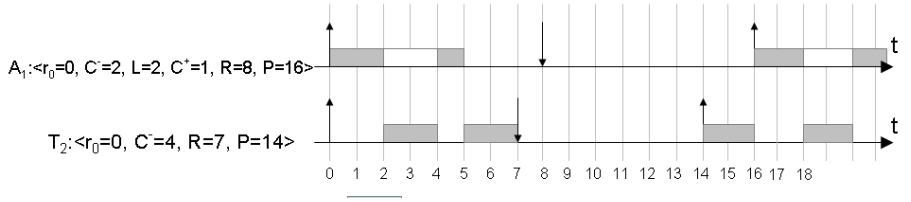


Figure 7.10 – Ordonnancement de tâches de traitement et d'acquisition avec $w_1 = -1$ et $w_2 = 4$

viseurs locaux (figure 7.11), il est en charge d'exécuter leurs décisions (sous-objectifs).

L'ordonnanceur se pose en tant qu'abstraction du niveau exécutif. En effet il est en charge de gérer toutes les contraintes (interférence capteur, précédence entre tâches, délai d'exécution etc..) liées à la mise en oeuvre d'un sous-objectif.

Son travail se divise en tâches (au sens travaux) :

1. calcul "on-line" d'un ordonnancement,
2. établissement des différents flux de données entre les modules,
3. activation des modules selon l'ordonnancement,
4. surveillance de l'activité des modules (retards, blocage).

Le calcul de l'ordonnancement a été détaillé dans la section 7.3, le fonctionnement et l'utilité des flux dynamiques dans une architecture ainsi que l'activation des modules ont été traités au chapitre 5. La surveillance de l'activité des modules est née d'un souci de robustesse face aux imprécisions des paramètres.

7.5.1 Adaptation des paramètres d'ordonnancement

En effet comme nous l'avons vu à la section 6.5 l'ordonnanceur calcule les dates d'activation de chaque module en fonction d'un certain nombre de paramètres (durée d'une tâche) estimés par l'utilisateur. Ces paramètres varient en fonction des capacités de l'architecture informatique matérielle (vitesse des processeurs etc..) et il est important de recaler leurs valeurs avec les valeurs observées. C'est l'ordonnanceur qui s'occupe de mesurer la durée de l'exécution des différents modules à chaque cycle et de reporter les valeurs pour les prochains calculs d'ordonnancement.

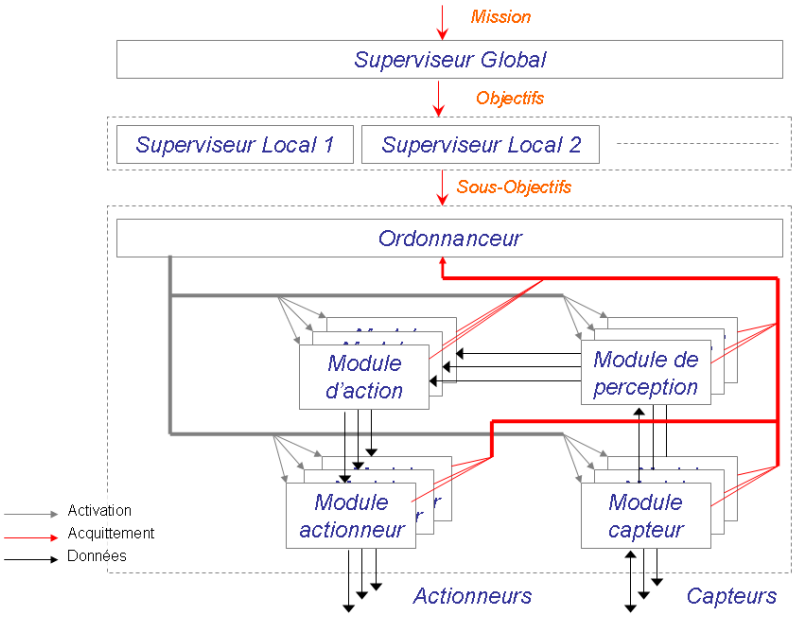


Figure 7.11 – Place de l'ordonnanceur dans l'architecture

Comme les valeurs mesurées (les durées) vont varier à chaque cycle, nous devons attribuer aux paramètres qui vont entrer dans le calcul d'ordonnancement une valeur qui est supérieure à la moyenne observée. En effet si pour chaque cycle on prenait comme valeur estimée la valeur observée au cycle précédent, on risque d'observer souvent des dépassements par rapport aux prévisions (figure 7.12.a). Si on donne la plus grande valeur observée, cela nous évite certes les dépassements mais cela nous empêche de nous adapter au cas où un module à une durée d'exécution qui va diminuer de façon significative et durable (exemple : le module fait des calculs plus simples pour le contexte donné) (figure 7.12.b).

La solution que nous avons retenue pour faire face aux évolutions des durées des tâches est de déterminer la durée d'une tâche en se basant sur la valeur moyenne des durées déjà observées et de l'écart moyen autour de cette valeur. Si nous supposons que :

- la durée d'exécution d'un module varie autour d'une moyenne M selon une loi gaussienne
- l'écart type des valeurs de durée observées est de σ

alors nous estimerons que la prochaine valeur de durée de la tâche vaudra M (unités de temps) et nous donnerons comme paramètre pour l'ordonnancement une durée de $M + 1.5\sigma$. En effet d'après les propriétés de la loi de répartition gaussienne, nous avons environ 87% de chance d'avoir une tâche dont la durée dépassera $M + 1.5\sigma$. Cela nous assure d'éviter les erreurs fréquentes de prévision et nous permet de palier au problème posé si la moyenne générale des durées venait à changer durablement.

Le nombre d'échantillons sur lequel ce calcul d'écart type se fait à une incidence particulière sur le temps de réaction aux variations durables des valeurs de durées. Plus le nombre d'échantillons est grand plus la réaction est lente. Dans notre cas nous choisissons d'effectuer les calculs sur une dizaine d'échantillons.

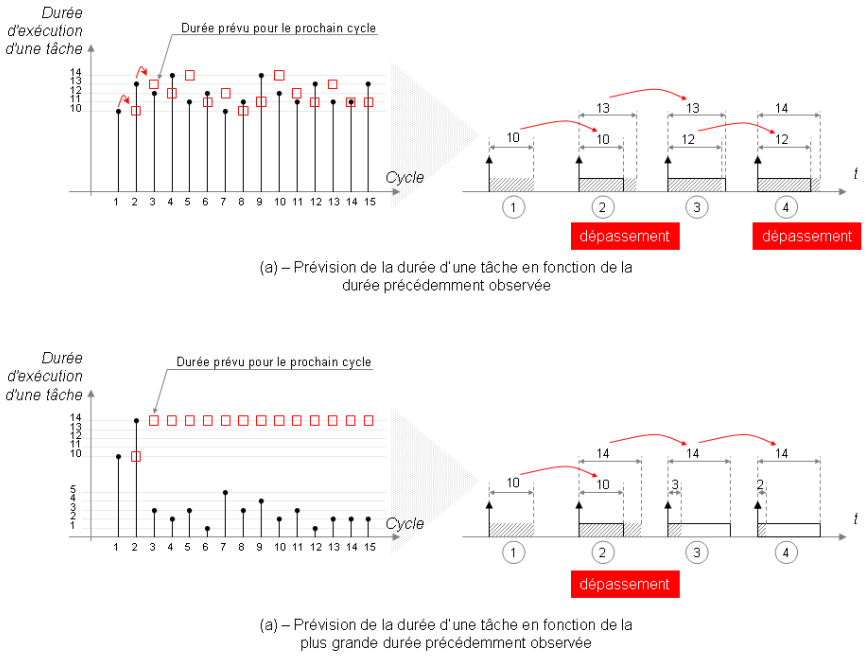


Figure 7.12 – Prévision des durées des tâches à chaque cycle

7.5.2 Réaction au blocage d'un module

Lorsque l'ordonnanceur active une tâche, il se met de suite en attente de l'acquittement de fin. Si la date de fin prévue de la tâche arrive avant que l'acquittement ne soit reçu alors on considère que la tâche est en retard. On passe alors directement à l'exécution des tâches qui la suivent. Les tâches qui étaient abonnées à des variables produites par la tâche en retard vont devoir travailler avec des données vieilles d'un cycle.

Au prochain cycle, l'ordonnanceur attendra encore la réponse de la tâche. La durée d'attente est alors en tout le double de celle initialement prévue. Si au delà de cette durée aucun acquittement n'est reçu, la tâche est considérée comme bloquée.

Lorsqu'une tâche se bloque, le sous-objectif dont elle représente un maillon, n'est plus exécutable. L'ordonnanceur arrête alors toutes les tâches relatives à ce sous-objectif puis produit un événement à destination du niveau décisionnel.

Même si la tâche n'est pas bloquée (i.e. elle a simplement duré plus que prévu au cycle considéré) cette erreur est prise en compte. En effet, les occurrences de ces situations de retard sont surveillées pour chaque module ; la règle de notification d'une erreur (signalée au niveau décisionnel) consiste à vérifier qu'un module ne dépasse pas plus de k fois la durée prévue, sur un horizon temporel glissant de n cycles ($k = 5$ et $n = 10$ dans la version actuellement implantée).

Résumé du chapitre 7 :

- L'exécution des modules bas niveau donne lieu à deux types distincts de tâches. Les tâches d'acquisition qui correspondent à l'exécution d'un module gérant une instrumentation acoustique et les tâches de traitement qui correspondent à l'exécution de tout autre module.
- Les tâches de traitement sont caractérisées par quatre paramètres temporels : r la date de réveil, c la durée de la tâche, R le delai critique et P la période. Ces tâches sont préemptibles.
- Les tâches d'acquisition sont caractérisées par six paramètres temporels : r la date de réveil, c^- la durée d'interrogation du capteur, L la latence de réponse du capteur, C^+ la durée de réception des données du capteur, R le delai critique et P la période. Ces tâches sont non-préemptibles.
- Nous proposons de calculer l'ordonnancement avec un algorithme en ligne à priorité variable. Cette priorité est calculée avec la formule : $p_i = w_1 * d_i + w2 * \lambda_i$ avec
  - d_i échéance de la tâche,
  - $\lambda_i = 0$ pour une tâche de traitement et $\lambda_i = 1$ pour une tâche d'acquisition,
  - $w_1 = -1$ et w_2 égale à la moyenne des durées des tâches de traitement.
- Nous adaptons en ligne les durées estimées des tâches en fonction des durées observées.

Troisième partie

Mise en oeuvre d'un contrôleur temps-réel d'un robot sous-marin

Introduction

Après avoir détaillé notre proposition en terme d'architecture logicielle pour les véhicules sous-marins autonomes, nous allons décrire un exemple d'implémentation de cette architecture. Ce travail a été effectué sur des robots sous-marins développés au Lirmm.

Nous présenterons dans un premier chapitre les véhicules de notre étude, notamment leur composition matérielle dont l'architecture matérielle du contrôleur présent à bord. Nous présenterons aussi les principales applications de ces véhicules.

Afin de déployer notre architecture logicielle, nous devons d'abord disposer d'un certain nombre de services informatiques offerts par un système d'exploitation à travers un *middleware* propre à notre architecture. Le système d'exploitation choisi doit en premier lieu offrir un fonctionnement temps-réel. Le choix et la conception de ces outils informatiques adaptés à notre cas d'étude seront détaillés dans un deuxième chapitre.

Dans un troisième chapitre nous exposerons les différents modules construits pour l'architecture logicielle d'un des véhicules sous-marins. Au moment de l'écriture de ce manuscrit, seul un des deux véhicules est opérationnel (au sens complètement assemblé et instrumenté).

Nous présenterons dans un quatrième chapitre les résultats obtenus durant les expérimentations menées en lac ; ces résultats peuvent être interprétés à partir du traçage daté de l'activité des modules ainsi que des variables échangées.

Chapitre 8

Architecture matérielle des véhicules sous-marins Taipan

8.1 Introduction

L'équipe de robotique sous-marine (RSM) du LIRMM a conclu plusieurs contrats relatifs à des opérations en mer, tels que l'étude des résurgences d'eau douce, la bathymétrie, l'inspection de zones côtières (détection de mines) et l'étude de la navigation en flottille. Pour mener à bien ces différentes missions, deux véhicules sous-marins ont été construits **Taipan II ou H160** et **Taipan 300**. Ces véhicules de type AUV (Autonomous Underwater Véhicle), sont équipés d'un ensemble de capteurs et actionneurs dont certains sont propres à leur navigation et d'autres dédiés à la mission qu'ils doivent accomplir.

L'architecture développée dans la partie précédente va être déployée sur les deux torpilles en remplacement d'une architecture de contrôle et de navigation jugée trop difficile à maintenir au regard des évolutions matérielles. D'autre part, l'une des torpilles, Taipan II, présente un ensemble de capteurs acoustiques qui peuvent potentiellement créer des interférences en cas d'utilisation simultanée et l'architecture proposée donne le moyen de palier à ces problèmes.

8.2 Les véhicules sous-marins Taipan

Les deux véhicules sous-marins développés au LIRMM, sont des véhicules de petite taille (environ 2 mètres) dédiés aux applications en eau peu profonde. Leurs petites dimensions et leurs poids relativement faibles (moins de 80 kg) permettent de réduire considérablement la logistique des différents essais en mer. Deux opérateurs avec un bateau de petite taille suffisent pour effectuer les opérations.

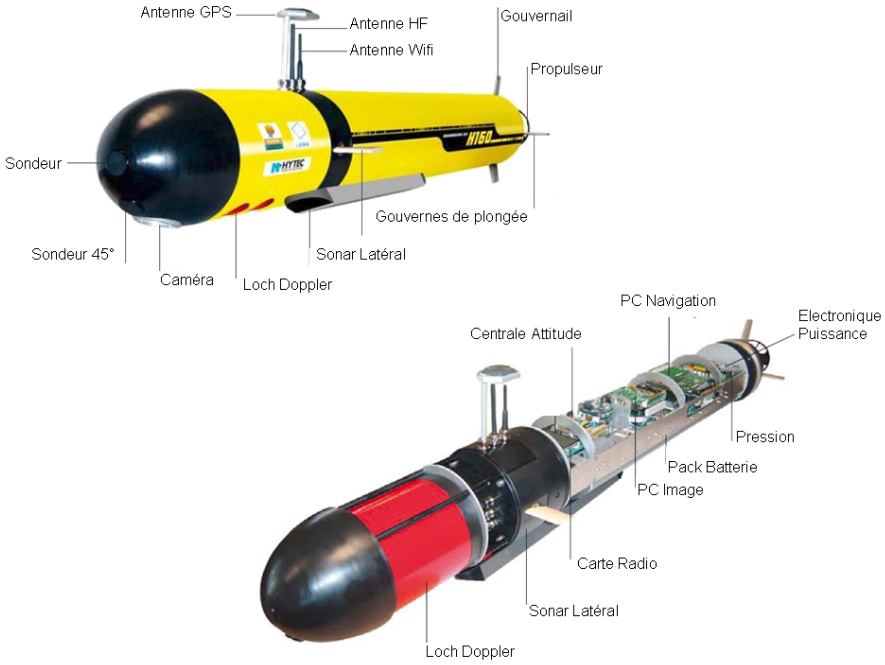


Figure 8.1 – Véhicule sous-marin autonome Taipan II ou H160

8.2.1 Taipan II

Le véhicule et son instrumentation

Le véhicule sous-marin autonome Taipan II [44][45] (voir figure 8.1) a une longueur de 1.80 m, un diamètre de 0.20 m. Ses caractéristiques sont :

- Poids : 55 kg,
- Profondeur maximum : 160 m,
- Vitesse moyenne : 3 noeuds,
- Autonomie : 4 heures.

Il embarque une instrumentation pour sa navigation propre, une instrumentation pour les besoins scientifiques de la mission et une instrumentation pour la communication, exposés en détails dans le tableau 8.1.

L'architecture informatique, au sens matériel, du H160 se compose d'une carte mère embarquée de petite dimension (PC104) muni d'un processeur Intel Celeron 400 Mhz. Cette carte permet de recevoir comme unité de stockage, un disque dur IDE et une carte compact flash. Elle dispose en outre de deux ports USB et deux ports série qui vont nous servir à connecter l'instrumentation de bord.

Les problèmes d'interférence

De nombreux capteurs en usage dans les véhicules marins et sous-marins sont basés sur la mesure des propriétés de propagation d'une onde acoustique dans l'eau. Dans la mesure où le médium utilisé reste le même (l'eau), les capteurs fonctionnant à des fréquences similaires (ou harmoniques) sont susceptibles de créer des interférences pouvant aller jusqu'à une invalidation complète du résultat issu du traitement des données qu'ils

Table 8.1 – Instrumentation de l'AUV H160

Communication			
Nom	Utilisation	Données brutes	Données traitées
Radio	Chargement de la mission, monitoring/téléopération en surface		
WiFi	Connexion à distance sur le PC embarqué (en surface et à courte distance)		
Modem acoustique	Monitoring en plongée et communication inter-véhicule (flottille)		
Pinger de secours	Balise de localisation		
Capteurs			
GPS	Localisation géoréférencée		
CTD	Mesure physico-chimique (caractérisation de sources d'eau douce)	Conductivité (mS.cm^{-1}), Température (°C), Pression (Pa).	Salinité ($g.l^{-1}$) obtenue à partir de la conductivité, température (°C), profondeur (m) obtenue à partir de la pression
Sondeur Acoustique 1 Sondeur Acoustique 2 Sondeur Acoustique 3	Suivi de fond et évitement d'obstacle	Temps de vol de l'onde acoustique	
Caméra vidéo	Acquisition d'image	Suivi d'amer	
Sonar à effet Doppler (Loch Doppler)	calcul de l'estime	vitesses (u,v,w) (m.s^{-1}) dans le repère local et attitude (ϕ, θ, ψ) ($rad.s^{-1}$) vitesses par rapport au fond ou vitesses par rapport à l'eau environnante	Attitude (ϕ, θ, ψ) (rad), vitesse ($\dot{x}, \dot{y}, \dot{z}$) ($m.s^{-1}$) obtenues à partir de (u,v,w et ψ) Position estimée de l'engin ($x, y, z, \phi, \theta, \psi$)
Sonar latéral	Cartographie, detection de pipeline, detection de mines	2 lignes perpendiculaires à l'engin constituées de mesures d'intensité de l'onde réfléchie (valeur entre 0 et 80 db) par le fond	
Centrale d'attitude et gyros.	Attitude et vitesse angulaire	Attitude (ϕ, θ, ψ) (rad), vitesse et accélérations angulaires (p,q,r) ($rad.s^{-1}$) et ($\dot{p}, \dot{q}, \dot{r}$) ($rad.s^{-1}$)	Attitude ϕ, θ, ψ (rad), vitesses et accélérations angulaires p,q,r ($rad.s^{-1}$)
Capteur de pression (6 bars)	Calcul de la profondeur (0-60m)	Pression (Pa)	profondeur (m)
Capteur de pression (16 bars)	Calcul de la profondeur (0-160m)	Pression (Pa)	profondeur (m)
Actionneurs			
Gouverne de cap (up)			
Gouverne de cap (down)			
Gouverne avant			
Gouverne arrière			
propulseur			

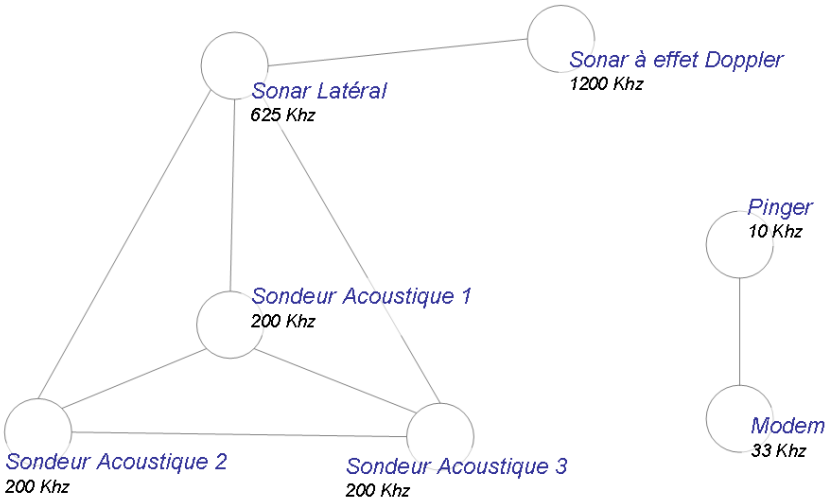


Figure 8.2 – Graphe des conflits de l'instrumentation acoustique de Taipan II

fournissent. Etant donné que ces données sont nécessaires à l'établissement de la navigation de l'engin, ce problème d'interférences doit être spécifiquement pris en compte, au risque de perdre l'engin.

Taipan II transporte sept instruments utilisant des ondes sonores (voir tableau 8.2).

D'après le tableau 8.2 il est évident que les trois sondeurs acoustiques entreront en conflit car ils utilisent la même fréquence pour émettre leur onde sonore. On peut aussi remarquer que le deuxième harmonique de l'onde émise par le Loch Doppler se situe à 600 kHz soit une fréquence très proche de celle du Sonar Latéral. On peut s'attendre alors à observer des erreurs de mesures lorsque ces deux capteurs fonctionnent simultanément. Le troisième harmonique du Sonar Latéral se situe dans la gamme de fréquence des ondes émises par les sondeurs acoustiques. Enfin, pour les mêmes raisons les données transmises par le modem acoustique peuvent être brouillées par le Pinger de secours.

Nous pouvons représenter plus simplement les interférences des capteurs par un graphe comme le montre la figure 8.3. Ce graphe servira pour la construction du graphe de compatibilité évoqué au chapitre 7. Ces présomptions (théoriques) d'interférences doivent être vérifiées par l'expérience car certains paramètres comme les positions relatives des capteurs ou les ondes convoyées par la coque ne peuvent être prévues.

8.2.2 Taipan 300

Le véhicule et son instrumentation

Le véhicule Taipan 300 est plus petit que Taipan II (voir figure 8.4). Avec un poids de 27 kg cet engin présente l'avantage d'être facilement manipulable. Le transport vers les sites d'expérimentation ne nécessite pas un matériel particulier.

Table 8.2 – Fréquences des ondes sonores utilisées par l'instrumentation de Taipan II

Instrument	*Fréquence*
Sonar à effet Doppler	1200 Khz
Sonar latéral	625 Khz
Sondeur Acoustique 1	200 Khz
Sondeur Acoustique 2	200 Khz
Sondeur Acoustique 3	200 Khz
Modem acoustique	33 Khz
Pinger de secours	10 Khz

Ses autres caractéristiques sont :
- longueur : 172 cm,
- diamètre : 15 cm,
- Profondeur maximum : 300 m,
- Vitesse moyenne : 2 $m.s^{-1}(3,9 noeuds)$,
- Autonomie : 2 heures.

Du fait de sa petite taille, Taipan 300 ne peut embarquer certains capteurs comme le Loch Doppler (utile pour connaître la vitesse de déplacement) ou le Sonar Latéral. Une section (ou tronçon) du véhicule située avant la tête permet de recevoir soit une caméra vidéo soit un capteur CTD soit un modem acoustique. On ne peut alors disposer à la fois de ces trois matériels sur le véhicule, on devra installer le matériel correspondant à l'application avant de commencer la mission. Le tableau 8.3 répertorie l'instrumentation présente actuellement sur ce véhicule.

L'architecture informatique, de Taipan 300 est identique à celle de Taipan II (figure 8.5) : une carte mère embarquée de petite dimension (PC104) munie d'un processeur Intel Celeron 400 Mhz. Cette carte permet de recevoir comme unité de stockage, un disque dur IDE et une carte compact flash. Elle dispose en outre de deux ports USB et deux ports série qui vont nous servir à connecter l'instrumentation de bord.

Le premier port série (COM1 ou ttyS0) est relié au module radio. Ce module radio établit une connexion avec un autre module identique relié au port série d'un ordinateur utilisateur. Ces modules sont configurés de façon à ce que la liaison radio ainsi établie entre les ports série des deux ordinateurs soit équivalente à un simple câble série croisé. Les communications radio étant impossible sous l'eau, ces modules sont exploités dans le cadre de téléopérations en surface.

Une carte électronique qui implémente un watchdog matériel est connectée sur le second port série de l'ordinateur embarqué. Ce watchdog requiert d'être stimulé à intervalles de temps réguliers de 1 seconde à défaut de quoi, il coupera automatiquement la puissance d'alimentation des moteurs du véhicule et connectera directement le GPS à la radio. Ainsi en cas de panne logicielle, les moteurs se mettront à l'arrêt et l'engin, fera naturellement surface du fait de sa flottabilité positive. Une fois à la surface l'opérateur pourra capter le signal radio qui contient la position du véhicule mesurée par le GPS.

Table 8.3 – Instrumentation de l'AUV Taipan 300

Communication			
Nom	Utilisation	Données brutes	Données traitées
Radio	Chargement de la mission, monitoring/téléopération en surface		
WiFi	Connexion à distance sur le PC embarqué (en surface et à courte distance)		
Modem acoustique	Monitoring en plongée et communication inter-véhicule (flottille)		
Pinger de secours	Balise de localisation		
Capteurs			
GPS	Localisation géoréférencée		
CTD	Mesure physico-chimique (caractérisation de sources d'eau douce)	Conductivité (mS.cm^{-1}), Température (°C), Pression (Pa).	Salinté (Siemens/m) obtenue à partir de la conductivité, température (°C), profondeur (z) (m) obtenue à partir de la pression
Sondeur Acoustique	Suivi de fond et évitement d'obstacle	Temps de vol de l'onde acoustique	
Caméra vidéo	Acquisition d'image		
Centrale inertielle	Calcul de l'estime pour une vitesse de croisière connue	Attitude (ϕ, θ, ψ) (rad), vitesse et accélérations angulaires (p,q,r) $(rad.s^{-1})$ et $(\dot{p}, \dot{q}, \dot{r})$ $(rad.s^{-1})$	Attitude ϕ, θ, ψ (rad), vitesses et accélérations angulaires p,q,r $(rad.s^{-1})$
Capteur de pression (1 bars)	Calcul de la profondeur (0-10m)	Pression (Pa)	profondeur (m)
Capteur de pression (10 bars)	Calcul de la profondeur (0-100m)	Pression (Pa)	profondeur (m)
Capteur entrée d'eau	detecter une fuite		
Actionneurs			
Gouverne de cap (up)			
Gouverne de cap (down)			
Gouverne avant			
Gouverne arrière			
propulseur			

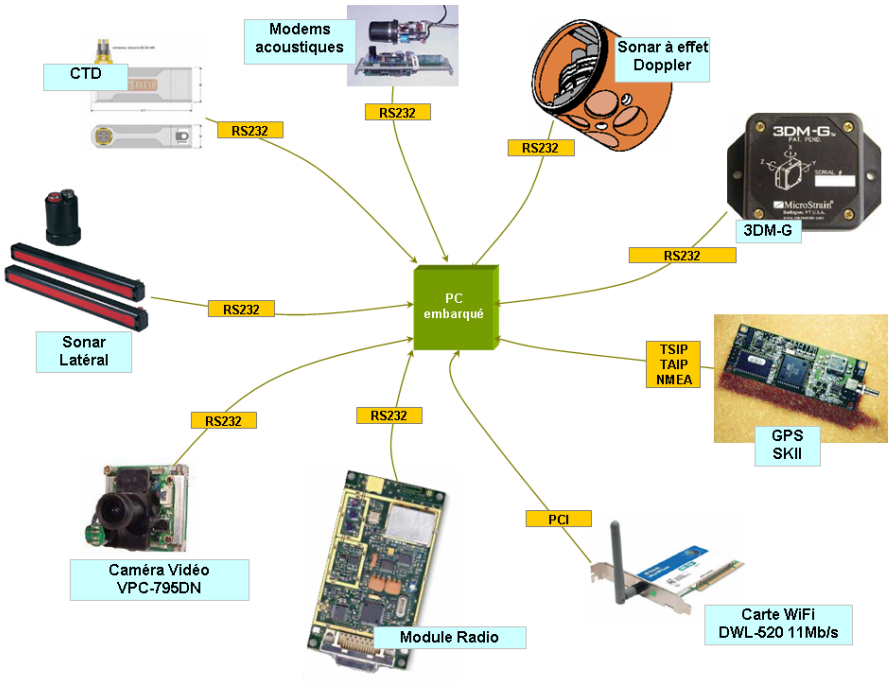


Figure 8.3 – Capteurs enbarqués sur Taipan II

Un capteur d'entrée d'eau surveille l'humidité à l'intérieur du véhicule et près des connexions étanches. Ce capteur est relié au PC 104 via une carte entrée/sortie analogique et signale la présence d'eau à l'intérieur de la coque. La réaction à une entrée d'eau sera purement logicielle (gestion de la remontée en surface).

Un dongle (ou clef) Wifi est connectée sur un des port usb et permet à la torpille de créer un réseau Ad-hoc. Ce réseau nous permettra de transférer des fichiers depuis un ordinateur vers le véhicule. Ce matériel sera utilisé pour les expérimentations en laboratoire ou sur site avant la mise à l'eau.

8.3 Applications mettant en oeuvre les véhicules

8.3.1 Taipan II

L'équipement scientifique de Taipan II lui permet d'être employé pour de nombreuses applications sous-marines. Nous citerons notamment l'inspection de pipeline, la remontée de source d'eau douce (sous-marine), la cartographie du fond marin.

Sur Taipan II, le Sonar Latéral est composé de deux barres longitudinales. Chacune de ces barres insonifie une ligne perpendiculaire à l'appareil et dirigée vers le bas. L'acquisition de ce capteur nous renseigne sur les altitudes relatives de chaque point de la ligne insonifiée. La juxtaposition de plusieurs lignes acquises durant le déplacement de l'engin représente alors une cartographie du fond à l'aplomb duquel le véhicule s'est déplacé. Avec un traitement d'image approprié, il est possible de détecter différents objets dont un pipeline dans le cas qui nous intéresse. L'"image" reconstituée à partir des données de ce capteur permettent d'une part de longer le pipeline étudié mais également d'analyser

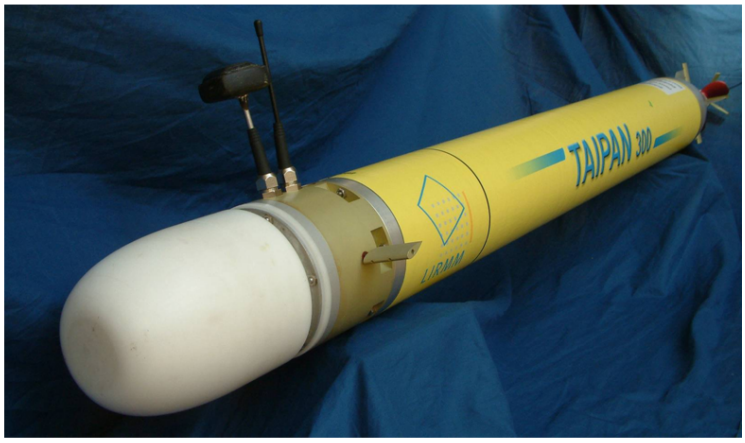


Figure 8.4 – Véhicule sous-marin autonome Taipan 300

sa surface à la recherche d'éventuelles fissures ou autres anomalies visibles à la surface. L'inspection de pipeline est une des principales applications de Taipan II.

Dans l'analyse de surfaces sur des images acoustiques fournies par le Sonar Latéral, l'information importante se trouve dans la différence relative des intensités des "pixels". Il existe cependant une relation de proportionnalité entre l'intensité de chaque pixel et l'altitude du point de l'environnement qu'il représente. En étalonnant le capteur, c'est à dire en déterminant le coefficient de la relation de proportionnalité, nous pouvons transformer les cartographies du Sonar Latéral en relevés bathymétriques qui constituent une des applications de Taipan II.

Un capteur CTD fixé sur la partie supérieure du véhicule, permet de mesurer trois grandeurs physiques dont la conductivité électrique de l'eau environnante. Cette conductivité est une fonction de la concentration de l'eau en sel. Ce type d'acquisition sera utilisé pour l'analyse des résurgences d'eau douce sous-marine. En effet l'eau douce ayant une faible concentration en sel par rapport à l'eau de mer, en relevant la salinité d'un volume d'eau donné, nous pouvons reconstruire le profil d'une source sous-marine. Ce profil n'est autre que la surface de transition de la salinité de l'eau de valeurs hautes aux valeurs basses.

La caméra embarquée permet d'acquérir des images du fond, mais aussi d'estimer le vecteur vitesse de l'engin par analyse du flot optique.

8.3.2 Taipan 300

Taipan 300 embarque très peu de matériel scientifique. La seule application envisageable aujourd'hui pour ce véhicule est la cartographie de source sous-marine d'eau douce. La procédure est la même que celle de Taipan II (voir section 8.3.1) et le capteur employé

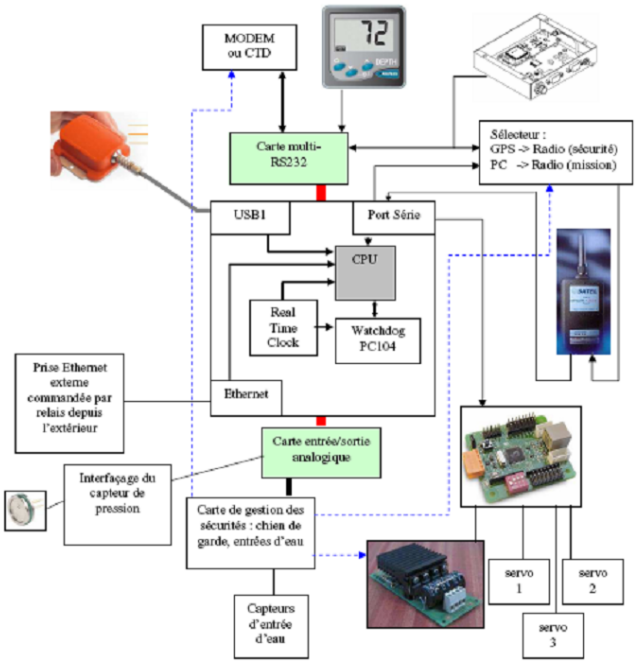


Figure 8.5 – Architecture matérielle informatique de taipan 300 et connexions avec l'instrumentation

dans ce cas est identique [46].

8.3.3 Taipan II et Taipan 300

Un des principaux problèmes rencontrés par les engins sous-marins (voir chapitre 1) provient de la très faible pénétration des ondes électromagnétiques dans l'eau. De ce fait tous les instruments de communication ou de mesure basés sur l'exploitation des propriétés de ces ondes deviennent inopérants sous l'eau dès que l'on dépasse le mètre de distance. Entre dans cette catégorie, le GPS, qui à partir de l'analyse des signaux émis par un certain nombre de satellites détermine sa propre position géographique (latitude, longitude et altitude). Le problème du positionnement en robotique sous-marine est partiellement résolu par l'utilisation de centrales inertielles. Embarquées à bord d'un robot sous-marin, ces centrales donnent des informations sur le déplacement de l'engin (accélération instantanée) qui permettent d'estimer le chemin parcouru depuis une certaine position (connue). Cependant les erreurs de mesures cumulées par ce type d'estimation ne permettent pas de disposer sur une longue période d'une estimation de position fiable.

Pour palier a ce problème, une des techniques envisagées dans la communauté robotique sous-marine est de combiner un véhicule de surface avec un véhicule de fond. Le premier véhicule effectue régulièrement des acquisitions GPS et transmet les informations à son homologue situé plus bas par l'intermédiaire d'une liaison acoustique. Cette technique met en exergue plusieurs problématiques liées notamment à la coordination des véhicules et à la maintenance du lien acoustique qui font par ailleurs l'objet de recherches scientifiques dans le domaine nommé coordination de flottille. Une des applications de la combinaison des deux véhicules Taipan II et Taipan 300 est d'offrir un banc d'expérimen-

tation pour les recherches menées au Lirmm dans ce domaine [47].

La communication acoustique employant elle aussi une instrumentation basée sur l'onde acoustique est exposée à des interférences venant de l'utilisation des autres capteurs acoustiques. Il sera alors nécessaire de la gérer au même titre que les autres capteurs.

8.4 L'aspect commande des véhicules

Classiquement la commande d'un véhicule sous-marin de type AUV est découplée entre les plans horizontaux et verticaux. Ainsi, sous l'hypothèse que le véhicule comporte une efficace compensation statique du roulis, les commandes dans le plan horizontal et le plan de plongée peuvent être indépendamment élaborées. La compensation statique du roulis est directement fonction de la "distance métacentrique", distance verticale entre le centre de masse et le centre de flottaison, laquelle, comme nous le verrons dans le chapitre 11 rapportant les tests effectués sur Taipan 300, est insuffisante et produit un couplage indésirable qui induit des perturbations sur la tenue des consignes. Toutefois, lors de l'élaboration de la commande, nous considérons ce phénomène de couplage comme négligeable. Ainsi, nous avons implémenté différents types de commande pour les plans horizontaux et verticaux.

- Une commande sur le principe du régime glissant est implémentée au sein des modules SMH (Sliding Mode Horizontal) et SMV (Sliding Mode Vertical). Cette commande non linéaire robuste permet de garantir un bon suivi des consignes malgré les incertitudes paramétriques relatives au modèle de l'engin.
- Nous avons choisi de réaliser les tests (rapportés au chapitre 11) avec une commande de type "PID guidé". Elle permet d'asservir l'engin à suivre la consigne de guidage, laquelle est élaborée en fonction d'une stratégie d'approche non linéaire permettant de borner la fonction d'erreur (par une fonction arctan). Ainsi, le réglage des gains de la commande est facilité par le fait que l'étage de commande est attaqué par une fonction bornée. Cette commande est implémentée au sein du module PIG (PID Guidé).

Pour tous ces modules, la période d'application de la commande est de 100ms.

Résumé du chapitre 8 :

- Le LIRMM dispose de deux véhicules sous-marins autonomes (AUV) : Taipan II et Taipan 300.
- Les deux véhicules ont une architecture informatique basée sur une carte processeur PC104 400 MHz.
- Taipan II embarque une instrumentation pouvant présenter des interférences.

Chapitre 9

Mise en place de l'infrastructure support de l'architecture logicielle

9.1 Introduction

Dans la partie 2 nous avons proposé une architecture logicielle pour les robots sous-marins. Cette architecture est construite sur une brique de base "le module". Ensuite nous avons proposé une construction possible d'architecture pour le domaine d'application qui nous intéresse.

Dans ce chapitre nous allons mettre en application la proposition du chapitre 5 concernant la construction d'un *middleware* sur lequel reposera la construction des modules de l'architecture logicielle. Puis nous développerons un fichier template de module qui va servir de souche à compléter pour obtenir un module particulier. Mais avant tout cela nous discuterons du système d'exploitation que nous allons choisir pour mettre en oeuvre tous ces développements sur l'architecture matérielle embarquée des AUVs Taipan.

9.2 Environnement de développement

9.2.1 Les systèmes d'exploitation

La figure 9.1 reprise du chapitre 5 montre les couches logicielles sur lesquelles va reposer notre architecture. Tout d'abord le système d'exploitation. D'une part il représente une abstraction du matériel, d'autre part il offre un certain nombre d'outils (compilateurs, outils de parsing etc...) qu'on exploitera dans la mise en oeuvre des modules de l'architecture.

Un système d'exploitation est généralement bâti autour d'un noyau central qui en est la partie fondamentale. Le noyau fournit des mécanismes d'abstraction du matériel, notamment de la mémoire, du (ou des) processeur(s) (lancement des programmes, ordonnancement, etc.), et des échanges d'informations entre processus et périphériques matériels.

Autour de ce noyau on trouve un ensemble d'éléments logiciels comme des pilotes, (éléments logiciels permettant de gérer les périphériques matériels) non supportés au

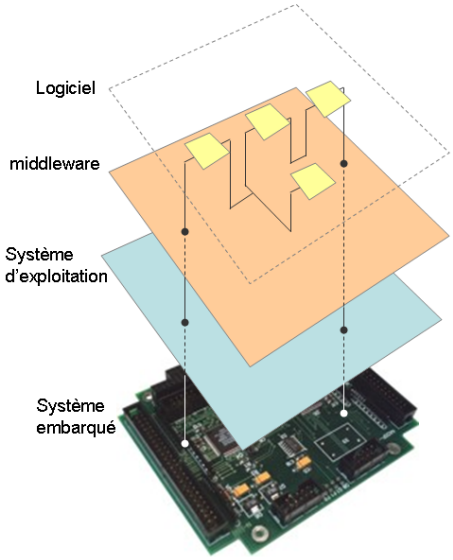


Figure 9.1 – Organisation des éléments logiciels et matériels d'un contrôleur

niveau du noyau. On y trouve aussi des outils destinés à l'utilisateur comme les terminaux de commande, les éditeurs de texte, les environnements de programmation etc.

Un système d'exploitation constituant la base sur laquelle les utilisateurs construisent leur travaux, il doit être le plus performant et le plus sûr possible. Les familles les plus connues de ces systèmes sont :

- les systèmes Microsoft Windows orientés vers le marché grand public, développés pour les architectures processeurs x86.
- les systèmes Mac OS, développés pour les machines Macintosh basées jusqu'à récemment sur des processeurs PowerPC.
- les systèmes GNU/linux, acceptant presque toutes les architectures processeur.

9.2.2 Le temps-réel

Il existe d'autres familles moins connues, chacune étant maintenue pour des applications précises, notamment le temps réel. Le temps réel est une faculté nécessaire dans notre application, notamment parce que notre logiciel doit être en mesure de produire des commandes à des dates précises pour contrôler de façon déterministe l'instrumentation du véhicule.

Parmi les systèmes d'exploitation temps réel, on trouve des systèmes purement dédiés à cette application et d'autres systèmes "hybridables", c'est à dire que l'on peut transformer en système temps réel.

systèmes d'exploitations temps-réels

- VxWorks
- μC-OS
- *QNX*

système d'exploitations "hybridables"

- Microsoft Windows avec le patch RTX
- GNU/Linux avec le patch RTAI
- GNU/Linux avec le patch RtLinux

Pour notre application nous avons choisi de travailler sous GNU/Linux patché avec RTAI. Le système linux a l'avantage d'être très répandu et a réuni une communauté active de développeurs et d'utilisateurs sur laquelle on peut s'appuyer. D'autre part il s'agit d'un système développé librement. RTAI à la différence de RtLinux est un système opensource (libre).

Linux est constitué d'un noyau central et d'un certain nombre d'outils périphériques. L'ensemble de ces outils détermine ce qu'on appelle une distribution linux, le noyau central étant la partie commune. En effet le noyau est développé dans le cadre du projet open-source "vanille" réunissant un ensemble de développeurs volontaires à travers le monde. Divers industriels reprennent et modifient plus ou moins ce noyau et le complètent avec leurs propres outils (environnements graphiques, pilotes de périphérique etc...) pour produire leur distribution de linux. Les distributions les plus connues sont :

- Redhat,
- Mandriva anciennement Mandrake,
- Debian

Ces distributions sont destinées au grand public et pour cela elles intègrent un grand nombre d'outils (traitements de texte, décodeurs vidéo, etc.) inutiles pour notre application et gourmands en place mémoire.

Il existe des distributions plus "légères" qu'on peut personnaliser à notre convenance :

- Slackware Linux (http ://www.slackware.com) est une distribution compacte,
- Linux From Scratch (LFS) (http ://www.linuxfromscratch.org) qui est plutôt une sorte de guide de construction d'un système d'exploitation complètement personnalisé depuis un noyau vanille.
- Gentoo Linux (http ://www.gentoo.org) basé sur la compilation locale de toutes les sources du système d'exploitation.
- ...

Ce qui nous intéresse dans ce travail est de maîtriser au maximum la composition de notre système d'exploitation pour ne mettre que l'essentiel. Gentoo Linux et Linux From Scratch présentent l'avantage de permettre de construire son propre système d'exploitation de façon incrémentale, libre à l'utilisateur d'omettre certains outils dont il n'a pas l'utilité. Le système construit dans ce cas est complètement compilé sur la machine hôte ce qui pousse ses performances au maximum.

Le principal inconvénient de ce genre de distribution est de nécessiter un temps de construction et installation important. A la différence de Linux From Scratch, Gentoo linux propose un certain nombre de paquets de base pré-compilés, prêts à l'installation. Cette technique réduit considérablement le temps de construction du système d'exploitation au détriment des performances mais cela reste encore un bon compromis performances/temps d'installation.

9.2.3 RTAI

RTAI (Real Time Application for Linux) est un ensemble d'outils permettant de faire du temps réel avec un système Linux traditionnel. Développé autour d'un projet open-source initié par l'Ecole Polytechnique de Milan (Politecnico di Milano), RTAI s'installe en deux phases :

- modification du noyau Linux de la distribution installée pour le rendre temps-réel,
- installation des outils offrant des mécanismes nécessaires pour développer des application temps réel (création de fifos, sémaphore, boîtes aux lettres, etc..)

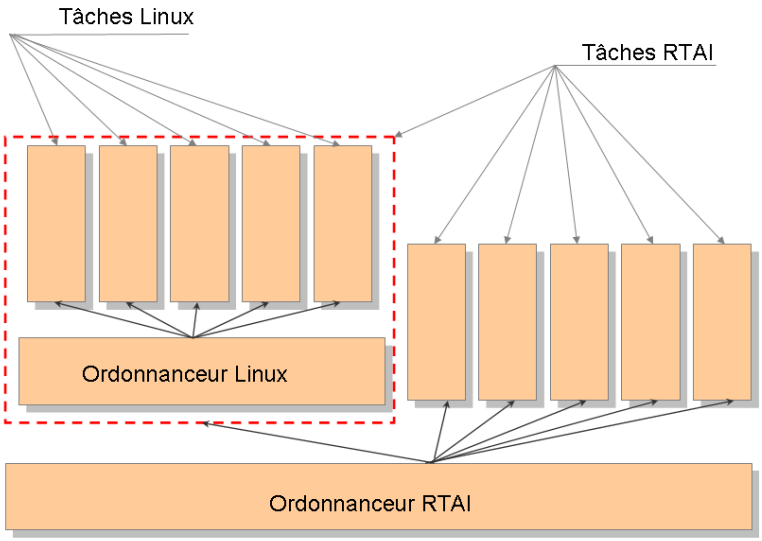


Figure 9.2 – Articulation entre l'ordonnanceur RTAI et l'ordonnanceur Linux

On dispose alors d'un système linux avec des capacités temps réel. Plus précisément, toutes les tâches linux y compris le noyau (comprenant un ordonnanceur) sont considérées comme une seule tâche de basse priorité ordonnancée par RTAI (figure 9.2).

A partir de la distribution Gentoo Linux, patchée RTAI, nous avons construit notre installation sur une carte compact flash de 8 G (plus résistante aux "chocs" que les disques durs).

9.3 Middleware

Le *middleware* qui va être utilisé dans ce travail a été implémenté sous forme d'une bibliothèque de fonctions (*libmodule*), développée en langage C durant la thèse et un stage de Master [48]. En utilisant ces fonctions lors de la construction de chaque module, on garantit d'avoir un comportement identique vis à vis des principaux mécanismes (communications, lancement/arrêt de module etc.) exploités au sein de l'architecture.

Comme décrit au chapitre 5, un module renferme une partie dite "système" offerte par le *middleware* et une partie dite "utilisateur" à la charge du développeur. La partie

"système" se charge des aspects communication entre modules (transferts de données, d'événements entre modules) ainsi que des aspects processus (initialisation, terminaison de la tâche, changement de priorité).

Pour construire un module, l'utilisateur remplit un fichier avec des paramètres de configuration (que nous spécifierons tout au long de cette étude) et une fonction (**moduleMain()**) qui représente le code spécifique de son module. Ce fichier est ensuite compilé avec la librairie *libmodule* pour donner un exécutable intégrant les aspects utilisateur et les aspects système susmentionnés.

Avant d'aborder les aspects communication, nous allons situer notre ordonnanceur par rapport à celui de RTAI.

9.3.1 L'ordonnanceur

Il faut bien distinguer l'ordonnanceur développé pour l'architecture logicielle et l'ordonnanceur du système d'exploitation. En effet, ces derniers ne se situent pas dans le même niveau d'abstraction. L'ordonnanceur de RTAI est en charge de lancer et arrêter toutes les tâches présentes sur le système sans tenir compte de l'application à laquelle elles appartiennent et selon des règles qui lui sont propres (e.g. la tâche de plus haut niveau de priorité est élue).

L'ordonnanceur de l'architecture utilise les services du précédent (e.g. changement de priorité des tâches) pour pouvoir obtenir un résultat. Ses décisions sont basées sur une analyse de données dépassant le seul cadre des processus en compétitivité pour une unité de calcul (prise en compte des relations particulières entre tâche : précédence, exclusion mutuelle, sous-configuration commune etc.).

De plus l'intervention au sein de l'ordonnanceur RTAI serait un obstacle à l'évolution et l'indépendance de la plateforme.

9.3.2 Choix des mécanismes de communication

Au sein de l'architecture nous devons pouvoir établir des échanges de données et d'événements entre chaque module. Les systèmes temps réel proposent plusieurs façons d'échanger des données entre tâches :

- mémoire partagée,
- files d'attente FIFO,
- boîtes aux lettres.

Le mécanisme de mémoire partagée suppose de réserver une portion de mémoire aux modules qui auront à échanger des données. Le module producteur de la donnée doit l'écrire dans une zone précise de cette mémoire et le module consommateur devra aller la chercher à cet endroit. Un des avantages de cette technique par rapport aux techniques basées sur l'envoi de message est que dans le cas d'un échange entre un producteur et plusieurs consommateurs, les consommateurs récupéreront à chaque lecture la dernière donnée produite (et écrite). Les problèmes liés à cette technique sont :

1. synchronisation nécessaire entre l'écrivain et le lecteur d'une donnée,
2. impossible à mettre en place sans briser la modularité.

De plus la création d'une mémoire partagée nécessite la connaissance de toutes les variables (taille de l'empreinte mémoire) qui vont être amenées à être échangées dans

l'architecture. Cette création doit être faite par une entité particulière de l'architecture. De plus chaque tâche lectrice d'une donnée doit connaître la zone dans cette mémoire partagée à laquelle elle se trouve. Pour connaître cette zone, elle doit être précisée au moment de l'implémentation du module ce qui fige du coup la relation entre ces modules et remet en question le principe de modularité au sein de l'architecture.

Les fifos et les boîtes aux lettres sont des mécanismes proches. Une boîte aux lettres est une portion de mémoire d'une taille déterminée, identifiée auprès du système temps réel par un nom. Les mécanismes offerts par le système temps réel permettent d'écrire et de lire dans cette boîte aux lettres des messages de n'importe quelle taille pourvu qu'elle ne dépasse pas sa capacité.

Par contre les mécanismes de lecture et d'écriture dans une fifo imposent de lire et d'écrire des messages de taille identique. La taille zone allouée à cette entité devant être un multiple de la taille d'un message. Une fifo ne peut alors pas par exemple recevoir des variables de tailles différentes.

Le principal inconvénient de ces entités d'échange de données basées sur l'envoi de messages est de dupliquer les données. Le module écrivain doit envoyer sa donnée dans la boîte aux lettres ou la fifo de chaque module consommateur de sa donnée. Cela peut commencer à poser problème lorsque les données sont très volumineuses (image vidéo par exemple... ce que l'on échange pas en temps-réel dans notre contexte).

Pour notre implémentation, nous avons retenu le mécanisme d'envoi de messages par boîte aux lettres, offrant plus de flexibilité (possibilité d'envoyer des messages de tailles différentes) que le mécanisme de fifos.

Initialisation et terminaison

Le point d'entrée principal d'un module (c'est à dire la fonction **main()** en langage C) est fournie par notre librairie, le code utilisateur n'étant qu'une fonction (**moduleMain()**) appelée par le programme principal. Ce programme principal contient les procédures d'initialisation et de terminaison du module.

Plusieurs choses sont faites pendant l'initialisation du module dont :
- création de la tâche temps réel,
- création des boîtes aux lettres.

Par convention lors de la création de la tâche nous lui attribuons un nom système composé du nom sur 3 lettres (un identifiant) donné par le développeur du module précédé de la lettre "T" comme tâche. RTAI nous permet de créer un certain nombre d'objets dont les tâches les boîtes aux lettres etc... et de les désigner par un nom composé de 6 lettres.

Le fichier rempli par l'utilisateur va contenir comme paramètres de configuration, la déclaration du nom du module. Ce nom va être utilisé pour la création de la tâche temps réel.

Dans les paramètres de configuration l'utilisateur doit inscrire pour chaque variable consommée par le module :
- adresse mémoire (locale) pour le stockage de la variable,
- nom du module producteur de la variable (dans le cas où ce dernier est connu avant la compilation cf. chapitre 5),
- port de production de la variable (si ce dernier est connu),
- périodicité de réception de la variable (en multiple entier de la période de production),

– taille de la variable.

Les adresses mémoire des variables vont permettre à la fonction de lecture des données (GetData()) de mettre à jour la variable. Le nom et le port du module producteur lorsqu'ils sont présents serviront à la fonction d'abonnement (subscribeUser) pour formuler auprès du module producteur de la donnée une demande d'abonnement à la périodicité indiquée. La taille de la variable servira à dimensionner les boîtes aux lettres pendant leur création. Le nom de chaque boîte aux lettres est composé du nom du module suivi du numéro de port sur 3 lettres le tout préfixé de la lettre "M" comme mailbox.

Pour chaque variable produite par le module on doit avoir :

– adresse (locale) de la variable,
– port de production,
– taille de la variable.

L'adresse permet d'identifier la variable, la taille est utilisée par les fonctions d'envoi de données (PostData()) pour déterminer la taille de la zone mémoire à transmettre à partir de l'adresse de la variable. Enfin, le port de production est l'identifiant par lequel la requête d'abonnement va designer la variable désirée par le module (demandeur).

Les déclarations se font de façon analogue pour les événements.

Enfin une boîte aux lettres dédiée aux requêtes est créée et nommée "MxxxRQ" où "xxx" est le nom donnée au module.

Lorsque toutes les initialisations ont été effectuées, le programme principal appelle la partie de code utilisateur, dont le squelette a été fourni (template). Une fois l'éxécution terminée (terminaison du module), le programme principal détruit toutes les boîtes aux lettres qu'il a créées et supprime la tâche temps réel.

Echanges bloquants/non-bloquants

Un module doit pouvoir s'articuler avec les autres par les communications. On distingue dans ces communications les flux de données et les flux de contrôle. Les flux de données concernent l'envoi et la réception de données et d'événements. Nous avons expliqué que ces communications ne doivent pas introduire de synchronisme entre les modules. Si durant son exécution, un module ne trouve pas de données dans une boîte aux lettres, il doit continuer son traitement en utilisant d'anciennes données. De même lorsque la boîte aux lettres de destination d'une donnée est pleine, le module continue son traitement sans la poster. Les trois fonctions suivantes ont été développées pour ce genre de communication :

– GetData() récupère une variable dans une boîte aux lettres,
– GetLastestData() pour récupérer le plus récent échantillon (i.e. dernier reçu)
– PostData() envoie une variable à tous les abonnés,
– PostEvent() envoie un événement à tous les abonnés.

L'ensemble de ces primitives assurent une datation automatique (transparente) des échanges.

L'activation d'un module par l'ordonnanceur se fait par envoi d'un message "Start" dans sa boîte aux lettres Requête. Lorsque le module a terminé un cycle d'exécution, il doit aller attendre la réception d'un nouvel ordre d'activation. La lecture dans ce cas est bloquante. La fonction correspondante est "waitForRequest()". Cette fonction est aussi sensible aux événements puisque ces derniers sont adressés dans les boîtes aux lettres Requête des modules.

Durant l'exécution, le module doit régulièrement surveiller la boîte aux lettres Requête pour réagir à la réception d'un ordre d'arrêt ou de suspension. Cette lecture assurée par la fonction "lookForRequest()" ne sera donc pas bloquante.

9.4 Templates

Comme illustré figure 9.3, la librairie *libmodule* met à la disposition du développeur un ensemble de services. Ces services vont concerner l'initialisation et la terminaison de la tâche et les différents échanges de données.

Un fichier type (Template) de construction d'un module est mis à la disposition du développeur. La structure de ce fichier sera plus amplement décrite au chapitre 10.

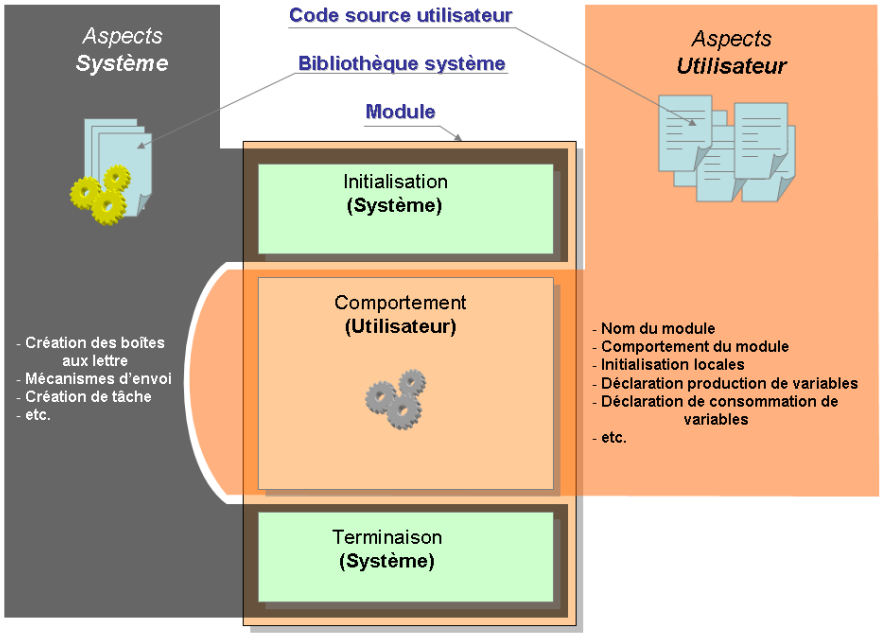


Figure 9.3 – Structure d'un module

9.5 Maintenance de la librairie de module

La librairie *libmodule* propose un ensemble de fonctions destinées à être utilisées dans la construction des modules. Pour l'instant implémentée en langage C, cette librairie est prévue pour être installée sous un système linux et utilisée avec RTAI. Elle regroupe une trentaine de fonctions et s'étend sur environ 3000 lignes de code.

Il est nécessaire de maintenir à jour une documentation de cette librairie, d'une part pour faciliter l'intervention de plusieurs développeurs et d'autre part pour permettre aux constructeurs de modules de connaître les fonctions mises à leur disposition. Nous avons mis en place pour cela le système de documentation Doxygen. Ce système permet de

générer automatiquement une documentation de la librairie à partir du code source commenté.

Résumé du chapitre 9 :

- Gentoo Linux/RTAI est le système d'exploitation temps-réel choisi comme support de notre architecture logicielle.
- L'implémentation de notre architecture sera supportée par le *middleware* "*libmodule*". *libmodule* est une librairie de fonctions en langage C fournissant tous les codes nécessaire au fonctionnement des modules : code d'initialisation, terminaison, échange de données/controle etc...
- Le mécanisme de boîte aux lettres est retenu pour supporter les différent flux au sein de l'architecture.

Chapitre 10

Construction du logiciel

10.1 Introduction

Dans les deux chapitres précédents, nous avons décrit la structure informatique (au sens matériel) des véhicules du LIRMM ainsi que les environnements logiciels qui vont supporter le développement de l'architecture logicielle pour ces robots. Les contrôleurs de ces robots sont centralisés autour de cartes processeurs de type PC104 sur lesquels nous avons installé Linux/RTAI ainsi que la librairie *libmodule*. Cette libraire va mettre à notre disposition un ensemble de fonctions qui vont être utilisées pour la construction des modules de chaque architecture logicielle.

Bien que le système Linux/RTAI autorise la programmation des tâches en d'autres langages comme le langage C++, certains mécanismes utilisés par ces langages ne sont pas garantis temps-réel.

Les modules que nous développerons sont destinés au véhicule Taipan 300, cependant comme l'architecture informatique (matérielle) est identique pour les deux engins nous pourrons réutiliser un certain nombre de modules. De plus les deux torpilles ayant un certain nombre d'instruments de bord identiques, un module gérant une instrumentation particulière sur l'un pourra être réutilisé sur l'autre pour remplir la même fonction. Taipan II n'étant pas entièrement assemblée mécaniquement, nous n'avons pas travaillé à la réalisation des modules de son architecture logicielle. Seul un module gérant le Sonar Latéral de cette torpille a été mis au point.

Les différents développements relatifs à l'architecture logicielle impliquant l'intervention régulière sur les systèmes robotiques, cela nécessitent la mise en place d'une logistique particulière pour en faciliter la réalisation et protéger le matériel. Cela sera détaillé dans une première section. Les deux sections suivantes traiteront de la mise au point des modules bas niveau, et des modules haut niveau. Enfin nous terminerons ce chapitre par l'explication de l'implémentation et l'utilisation des bases de données "vocabulaire" spécifiées au chapitre 6.

10.2 Environnement de travail

10.2.1 Les machines

La construction d'une architecture logicielle requiert d'une part l'implication de personnes de compétences différentes et d'autre part l'intervention sur du matériel fragile.

L'intervention sur le robot nécessite de garder la coque de protection ouverte et de recharger les batteries. En plus des risques de détérioration de matériel que ces opérations comportent (surtout pour des non électroniciens), il est très peu aisé de développer sur le site de construction ou de transporter l'engin sur le site de travail.

Une solution aurait été de permettre à chaque intervenant de pouvoir développer sur son propre ordinateur tout le code ne nécessitant pas d'utiliser le matériel puis de rassembler le tout pour un test sur la machine. Cependant il n'est pas envisageable que chaque intervenant potentiel ait Linux/RTAI installé sur sa machine, ni qu'il ait l'expertise requise au niveau système pour mener à bien les développements, les configurations ou les mises à jour.

La solution retenue a été de centraliser les développements sur un serveur commun, appelé "*Perle*" accessible directement sur le réseau informatique du LIRMM. Sur cette machine, nous avons installé Gentoo Linux/RTAI de façon à obtenir une configuration identique au PC embarqué sur les véhicules sous-marins.

Les moyens les plus simples pour modifier ou copier des fichiers d'un ordinateur à un autre sur un même réseau sont ceux basés sur les systèmes de fichier réseau. Parmi ceux-ci nous avons choisi d'utiliser le système de fichier samba. Nous avons installé un serveur samba sur *Perle* qui met en ligne deux répertoires. L'un contenant l'arborescence de l'architecture logicielle et l'autre contenant les sauvegardes successives de cette arborescence.

Pour tester les développements, par exemple pour compiler et exécuter un module, il suffit de se connecter sur *Perle* en ssh (mode de connexion sécurisé) et exécuter les commandes shell adéquates.

De plus certains matériels (e.g. centrale inertielle) qui peuvent être extraits facilement du véhicule peuvent être connectés sur *Perle* pendant les développements.

Enfin sur le PC104 embarqué sur la torpille, nous avons également installé un serveur samba ainsi que le driver du système de fichier samba. Lorsqu'on doit faire un test complet ou lorsqu'on doit essayer un matériel non démontable, nous pouvons procéder de deux manières : soit on monte l'arborescence de l'architecture dans un répertoire sur le PC104 embarqué, soit on connecte le PC104 au réseau et on copie l'architecture logicielle depuis le serveur commun vers l'ordinateur du robot. La première solution permet de travailler sur les fichiers situés sur le serveur commun, c'est à dire qu'une modification de code sur le PC104 est automatiquement répercutée sur le source commun (fichiers du serveur *Perle*). L'inconvénient de cette méthode est la nécessité d'être toujours connecté au réseau et les temps de lecture/écriture sont plus grands. La seconde solution dissocie complètement les deux machines mais après l'opération il faut reporter sur les fichiers sources (situés sur *Perle*) les modifications faites sur le PC104. La première solution est utilisée pour des développements ne nécessitant pas de performance temporelle comme le test de l'instrumentation alors que le second est plus adapté pour des tests complets de l'architecture logicielle sur le véhicule.

10.2.2 Travail en commun

Nous avons centralisé sur *Perle* un ensemble d'outils d'aide au développement comme des commandes d'activation/arrêt de module, de calcul de temps d'exécution de module etc. Nous y maintenons installée la dernière version de la librairie de module avec sa documentation *Doxygen*. Cette documentation nous permet de naviguer facilement dans le code source.

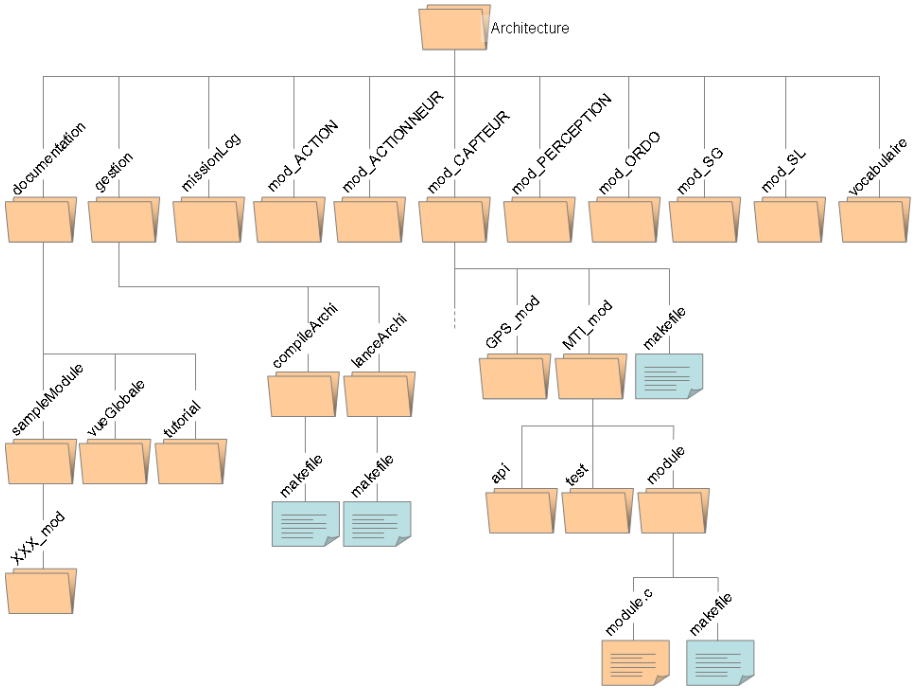


Figure 10.1 – Vue partielle de l'arborescence de l'architecture logicielle

L'arborescence de l'architecture (figure 10.1), permet de structurer les développements. Pour construire un module, on recopie simplement le dossier "XXX_mod" dans le dossier de la classe de module correspondante. Nous avons quatre classes de modules :

- modules capteurs : gèrent les capteurs,
- modules perception : traitements des données produites par un ou plusieurs modules capteur, fusion de données, estimateurs, observateurs, etc.
- modules action : implémentent les lois de commande,
- modules actionneurs : gèrent les actionneurs.

Le dossier XXX_mod doit être renommé en remplaçant "XXX" par le nom du module sur trois lettres. Ce dossier contient un fichier template de module que nous expliquerons par la suite et un fichier contenant les règles de compilation pour ce fichier pour la génération du module.

La copie de ce dossier suffit à l'inscrire dans la liste des prochaines compilations générales de l'architecture. Cette compilation générale est obtenue par l'exécution du fichier de règles du dossier "compilArchi".

10.2.3 Template de module

Comme nous l'avons évoqué au chapitre précédent, la librairie *libmodule* offre un certain nombre de fonctions qui vont être utilisées lors de la réalisation d'un module. Certaines fonctions seront toujours appelées au même endroit dans tous les fichiers (e.g. fonction d'initialisation etc..) qui vont implémenter un module. Il en est de même pour les paramètres à déclarer, un certain nombre d'entre eux reviendront systématiquement dans tous les modules. Les autres fonctions et paramètres seront appelés à partir du code de l'utilisateur.

Pour faciliter la mise en oeuvre d'un module, nous mettons à disposition du développeur un squelette de module contenant tous les appels communs aux fonctions de *libmodule* organisés de façon à faire un module "vide" présentant tous les mécanismes nécessaires dont l'initialisation et la terminaison du module, l'envoi et la réception de données, etc. Pour constituer son propre module, l'utilisateur n'aura qu'à compléter ce template par des informations spécifiques à l'utilisation qu'il va en faire.

La structure du template est la suivante :

Déclaration des variables échangées

fichier template section 1

```
1 :  // Inclusion des bibliotèques
2 :   #include <libmodule/libmodule.h>
3 :   #include "api.h"
4 :  // — ———————— —
5 :   MODULE_AUTHOR("---");
7 :   MODULE_DESCRIPTION("---");
8 :  // Declaration de Variables
9 :  // — ———————— —
10 :  // Nom du module
11 :   char MODULE_NAME[] = "---";
12 :  // { &Variable, "Nom du module", Port #, Periodicite, Taille }
13 :   ModuleUse IUSE[] = {{&---, "---", ---, ---, sizeof(---) } , IUSE_TERM };
14 :  // { &Variable, Port #, Taille }
15 :   ModuleProduce IPRODUCE[] = {{&---, ---, sizeof(---)}, IPRODUCE_TERM };
16 :  // { &Variable, "nomDuModule", Port #, Taille }
17 :   ModuleReact IREACT [] = {{&---,"---", ---, sizeof(---)}, IREACT_TERM };
18 :  // { &variable, Port #, Taille }
19 :   ModuleDetect IDETECT [] = {{&---, ---, sizeof(---)}, IDETECT_TERM };
20 :  // Taille des variable param recue :
21 :   size_t PARAM_SIZE[] = { sizeof(---), sizeof(---), PARAM_SIZE_TERM };
```

Les tirets dans le code doivent être remplacés par les informations concernant le module tel que précisé par les commentaires. Le tableau `IUSE` peut contenir plusieurs sous-tableaux, chaque sous-tableau contient les informations concernant une des variables consommée par le module :

- adresse de stockage (mémoire) de la variable consommée,
- nom du module producteur de la variable,

– port de production de la variable,
– périodicité de réception (multiple entier de la période de production),
– taille de la zone de stockage de la variable.

On ajoute dans ce tableau autant de sous-tableaux que le module possède de variables consommées. Les tableaux suivants se remplissent de manière analogue et concernent respectivement : les variables produites, les événements consommés et les événements produits. Le dernier tableau doit contenir les tailles des variables reçues en paramètre ou par événement.

Tous ces tableaux vont servir aux différentes fonctions d'initialisation du module pour pouvoir dimensionner les boîtes aux lettres ou pour effectuer les abonnements nécessaires auprès des modules concernés.

fichier template section 2 simplifiée

```
22 :   int ModuleMain(int argc, const char * argv[])
23 :   {
24 :     traitement_init();
25 :     MessageRecu.Type=REQ_UNKNOWN;
26 :     while(!FIN)
27 :     {
28 :       switch(MessageRecu.Type)
29 :       {
30 :         case REQ_UNKNOWN :
31 :           WaitForRequest(&MessageRecu);
32 :           break;
33 :
34 :         case REQ_START:
35 :           while(!interact)
36 :           {
37 :             switch(etat)
38 :             {
39 :               case 0:
40 :                 traitement_1();
41 :                 etat = 1;
42 :                 break;
43 :               case 1:
44 :                 traitement_2();
45 :                 etat = 0;
46 :                 cycleEnCours = 0; interact = 1; MessageRecu.Type=REQ_UNKNOWN;
47 :                 SendAck( MessageRecu.Param.Start.ModAck, MessageRecu.Param.Start.WantedAck,  ACK_FINISHED, MessageRecu.Param.Start.AckID  );
48 :                 break;
49 :             }
50 :             if(cycleEnCours)
51 :             {
52 :               ret=LookForRequest(&MessageRecu);
53 :               if (ret!= NULL) interact=1;
54 :             }
55 :           }
56 :           break;
57 :     ....
58 :     ....
59 :         case REQ_STOP:
60 :           etat = 0;
61 :           MessageRecu.Type=REQ_UNKNOWN;
62 :           break;
63 :
64 :         case REQ_KILL:
65 :           FIN = 1;
66 :           MessageRecu.Type=REQ_UNKNOWN;
67 :           break;
68 :       }
69 :     }
71 :     traitement_exit();
72 :     return 0;
73 :   }
74 :   void traitement_init() { /* CODE UTILISATEUR INITIALISATION DU MODULE */}
75 :   void traitement_exit() { /* CODE UTILISATEUR TERMINAISON DU MODULE */}
76 :   void traitement_1() { /* CODE UTILISATEUR PORTION 1 */}
77 :   void traitement_2() { /* CODE UTILISATEUR PORTION 2 */}
```

La suite du fichier template dont nous avons présenté une version simplifiée ci-dessus, commence par la fonction "ModuleMain()" que nous avons évoquée dans le chapitre précédent. Cette fonction sera appelée par une fonction main(), (point d'entrée principal du programme représentant le module) contenue dans *libmodule*. Dans ModuleMain() nous implémentons le réseau de Petri présenté à la section 5.5. Le module attend un message sur sa boîte aux lettres Requête (cf. ligne 31) et à la réception d'une requête, il la traite en fonction de sa nature (RQ_START ligne 34, RQ_STOP ligne 59, RQ_KILL ligne

64). La requête RQ_START correspond à la demande d'exécution du code utilisateur (ligne 34 à 56). Comme nous l'avons expliqué, ce dernier doit être découpé en plusieurs étapes (ligne 40 et 44) pour assurer un certaine réactivité au module. Entre l'exécution de chaque étape de code utilisateur, la boîte aux lettres Requête est consultée (ligne 52). Après avoir fini l'exécution du code utilisateur, le module se remet de nouveau en attente d'une requête.

Le code relatif à l'utilisation spécifique du module construit sur cette base sera ajouté dans les fonctions :

- traitement_init() : code utilisateur d'initialisation du module,
- traitement_exit() : code utilisateur de terminaison du module,
- traitement_i() : $i^{\text{ième}}$ portion de code utilisateur.

Les fonctions traitement_init() et traitement_exit() contiennent des codes spécifiques au module construit et doivent être distingués des initialisations et terminaisons faites par la fonction main() contenue dans *libmodule* et englobant la fonction ModuleMain() :

point d'entrée principal et point d'entrée utiliateur simplifiée

```
1 : int main(int argc, const char * argv[]) // point d'entrée principal
2 : {
3 :   ....// PROCEDURES D'INITIALISATION DU MODULE
4 : // ModuleMain(argc, argv);  // point d'entrée utilisateur
5 :   .... // PROCEDURES DE TERMINAISON DU MODULE
6 : }
```

Les fonctions traitement_i() seront complétées par l'utilisateur. Elles ne doivent pas contenir d'appel système ni de boucle infinie. Un certain nombre de fonctions sont fournies pour envoyer et recevoir des données et des événements : GetData(), GetLastestData(), PostData(), PostEvt().

10.3 Les modules du bas niveau

Nous avons classifié les modules bas niveau en quatre catégories :

- modules capteurs : gèrent les moyens de perception du robot,
- modules perception : effectuent des traitements sur les données provenant des modules capteur (e.g. fusion de données),
- modules action : implémentent les lois de commande,
- modules actionneurs : gèrent les moyens d'action du robot.

Cette classification est tout simplement structurante, elle n'a pas d'implication directe sur le mode de construction ou de fonctionnement des modules.

10.3.1 Modules capteurs

CTD

Le module CTD gère le capteur CTD (mesure de la conductivité, de la température et de la profondeur). A chaque activation, il va lire sur le port série une trame. Après décodage, il produit les valeurs suivantes :

Port	*Données produites*	*Type*	*unité*
0	conductivité	double	$mS.cm^{-1}$
1	température	double	$^{\circ}C$
2	pression	double	Pa

DIO

Le module DIO gère une carte externe sur laquelle on retrouve le watchdog et le capteur d'entrée d'eau. Pour que le watchdog reste inactif, ce module doit être activé au moins une fois par seconde. Il produit l'événement suivant :

Port	*Evènement produit*	*Type*	*unité*
0	entrée d'eau	int	

GPS

Le GPS est connecté via un câble convertisseur série/usb à l'ordinateur embarqué. Le module GPS récupère les trames GPS sous le format standard NMEA qu'il décode pour produire une donnée appelée GPSData :

Port	*Donnée produite*	*Type*	*unité*
0	données GPS	struct{ double latitude ;	m
		char latitudeC ;	direction cardinale
		double longitude ;	m
		char longitudeC ;	direction cardinale
		double altitude ;	m
		char altitudeC ;	
		double velociteKMH ;	$Km.h^{-1}$
		unsigned long UTCtime ;	s
		char quality ;	
		char nbofsatellite ;	
		double var ;}	

MTI

La centrale inertielle envoie des données à une fréquence de 100Hz via un câble convertisseur série/usb (ftdi). Le driver qui gère ce convertisseur sous linux, possède une fifo de 512 octets. La taille des paquets envoyés par la centrale étaient de 54 octets. Nous récupérons une donnée toutes les 100 ms, pour réduire le temps d'effacement de la fifo nous avons du modifier la taille de celle-ci (passage de 512 octets à 128 octets) en modifiant les fichiers source du driver ftdi. La durée d'exécution du module est passée d'environ 80 ms à environ 10 ms.

Port	*Données produites*	*Type*	*unité*
0	Orientation	struct{ float roll ;	*rad*
		float pitch ;	*rad*
		float yaw ;}	*rad*
1	Dynamique du véhicule	struct{float accX ;	$m.s^{-2}$
		float accY ;	$m.s^{-2}$
		float accZ ;	$m.s^{-2}$
		float gyrX ;	$rad.s^{-1}$
		float gyrY ;	$rad.s^{-1}$
		float gyrZ ;	$rad.s^{-1}$
		float magX ;	*mGauss*
		float magY ;	*mGauss*
		float magZ ;}	*mGauss*

PRF

Les capteurs de pression présents sur Taipan 300 fournissent une grandeur analogique représentant la pression du milieu ambiant, et permettent grâce à cela de mesurer la profondeur de l'engin sous l'eau. Taipan 3 possède 2 capteurs de pression, mais seulement un est utilisé actuellement. Il s'agit du capteur mesurant la profondeur entre 0 et -10 mètres. Le PC104 n'étant pas équipé d'entrées analogiques, une carte supplémentaire connectée à ce dernier permet de transformer le signal du capteur en une valeur numérique. Le module doit d'abord la configurer (tensions admissibles etc.) à l'initialisation avant de pouvoir l'interroger.

Port	*Donnée produite*	*Type*	*unité*
0	Profondeur	double	*m*
Port	*Evenement produit*	*Type*	*unité*
0	surface	int	

SON

Le module SON gère un sondeur placé à la tête de Taipan 300. Il permet de connaître à chaque instant la distance au fond.

Port	*Donnée produite*	*Type*	*unité*
0	Distance	double	*m*

RDI

Le module RDI est un module développé pour le sondeur à effet doppler non présent parmi l'instrumentation de Taipan 300, il est destiné à être implémenté sur l'architecture logicielle de Taipan II.

Port	*Données produites*	*Type*	*unité*
0	Vitesse linéaire par rapport au Nord	double	$m.s{-}1$
1	Vitesse linéaire par rapport à l'Est	double	$m.s{-}1$
2	Vitesse linéaire du véhicule	double	$m.s{-}1$
3	Roulis	double	rad
4	Tangage	double	rad
5	Lacet	double	rad
6	Portée	double	m
7	date_cs	long	$\frac{1}{10}s$

STC

Un simulateur a été réalisé pour pouvoir tester l'architecture de contrôle des AUV ainsi que pour régler facilement les paramètres des lois de commande. Ce simulateur détaillé dans [49] et [50], se connecte via une socket à l'architecture logicielle située sur un autre ordinateur (ordinateur embarqué par exemple). Il envoie les données capteur simulées et récupère les commandes à appliquer aux actionneurs. Pour pouvoir s'interfacer facilement avec ce simulateur, le module STC récupère les données capteurs simulées et les met à la disposition de tous les modules de l'architecture et un autre module ATS que nous verrons par la suite récupère les commandes destinées aux actionneurs pour les envoyer via une socket au simulateur.

Port	*Données produites*	*Type*	*unité*	*capteur emulé*
0	Données GPS	cf. module GPS port 0		GPS
1	Orientations	cf. module MTI port 0		Centrale d'attitude
2	Dynamique du véhicule	cf. module MTI port 0		Centrale d'attitude
3	profondeur	double		capteur de pression
4	conductivité	double	$S.m{-}1$	capteur CTD
5	température	double	$^{\circ}C$	capteur CTD
6	pression	double	Pa	capteur CTD
7	distance	double	m	sondeur acoustique

10.3.2 Modules perception

NAV

Le module de navigation (NAV) a pour rôle de calculer les positions, vitesses et direction du véhicule, d'après les mesures des capteurs. Il s'agit principalement d'un filtre de fusion de données, intégrant les mesures de vitesses et d'accélérations de façon à estimer la position absolue du véhicule. Cette estime intégre bien sûr les erreurs de mesures, provocant ainsi une dérive dans l'estimation de la position. Cette dérive, lorsqu'elle dépasse un seuil donné produit l'événement "estime décalée". La réaction à ce type d'événement est généralement un retour en surface de façon à acquérir une mesure de la position absolue du système par le GPS.

Port	*Données produites*	*Type*	*unité*
0	Vecteur état	struct { double x_North ;	m
		double y_East	m
		double z	m
		double phi	rad
		double theta	rad
		double psi	rad
		double u	$m.s^{-1}$
		double v	$m.s^{-1}$
		double w	$m.s^{-1}$
		double p	$rad.s^{-1}$
		double q	$rad.s^{-1}$
		double r	$rad.s^{-1}$
Port	*Données reçues*	*Type*	*unité*
0	données GPS	cf. module GPS port 0	
1	profondeur	double	m
2	orientations	cf. module MTI port 0	
3	Dynamique du véhicule	cf. module MTI port 1	
Port	*Evènement produit*	*Type*	*unité*
0	Estime décalée	struct { double date	ns
		double datePrec	ns
		double X_North_Nav	m
		double Y_East_Nav	m

10.3.3 Modules action

GSV

GSV est le module guidage "suiveur" c'est à dire qu'il implémente un guidage (au sens automatique du terme) très simple qui consiste à recopier simplement les consignes. A chacune de ses activations, il produit la valeur de consigne qu'il a reçue en paramètre lors de sa première activation.

Port	*Donnée reçue en paramètre*	*Type*	*unité*
0	Valeurs Désirées	struct { double u	$m.s{-}1$
		double theta	deg
		double cap	deg
		double z	m
		double reg_Mot	%
Port	*Donnée produite*	*Type*	*unité*
0	Valeurs Désirées	struct { double u	$m.s{-}1$
		double theta	rad
		double cap	rad
		double z	m
		double reg_Mot	%

WPT

Le guidage de type "waypoint" permet d'obtenir une trajectoire désirée en définissant les points de passage (waypoint). Ainsi, le module WPT produit les consignes à suivre pour rejoindre le waypoint i. Lorsque l'engin se trouve à une distance inférieure à un certain seuil (défini à l'intérieur du module) du waypoint i, l'événement "point de passage atteint" est produit.

Port	*Données reçues en paramètres*	*Type*	*unité*
0	valeurs Désirées	cf module GSV port 0	
1	Position désirée	struct { double x	m
		double y	m
		double z }	m
Port	*Donnée consommée*	*Type*	*unité*
0	vecteur etat	cf. module NAV port 0	
Port	*Donnée produite*	*Type*	*unité*
0	valeurs désirées	cf. module GSV port 0	
Port	*Evènement produit*	*Type*	*unité*
0	point de passage atteint	int	

PIH

Le module "PIH" implémente un contrôle linéaire de type "PID" dans le plan horizontal. Ainsi, il reçoit la référence et la mesure en cap et produit la consigne à appliquer à l'actionneur, la gouverne verticale.

Port	*Données consommées*	*Type*	*unité*
0	vecteur état	cf. module NAV port 0	
1	valeurs Désirées	cf. module GSV port 0	
Port	*donnée produite*	*Type*	*unité*
0	angle des gouvernes de cap	double	rad

PIG

Le module "PIG" implémente un contrôle de type "PID" assurant le suivi d'un guidage non linéaire. Il reçoit les mesures et les références à suivre pour produire les consignes des actionneurs horizontal et verticaux.

Port	*Données consommées*	*Type*	*unité*
0	vecteur état	cf. module NAV port 0	
1	valeurs Désirées	cf. module GSV port 0	
Port	*Données produites*	*Type*	*unité*
0	angle des gouvernes avant	double	rad
0	angle des gouvernes arrière	double	rad
0	angle des gouvernes de cap	double	rad

SMH

Le module "SMH" implémente un contrôle de type régime glissant (sliding mode) [47] dans le plan horizontal. Ainsi, il reçoit la référence et la mesure en cap et produit la consigne à appliquer à l'actionneur, la gouverne verticale.

Port	*Données consommées*	*Type*	*unité*
0	vecteur etat	cf. module NAV port 0	
1	valeurs Désirées	cf. module GSV port 0	
Port	*Données produites*	*Type*	*unité*
0	angle gouverne de cap supérieure	double	*rad*
1	angle gouverne de cap inférieure	double	*rad*

SMV

Le module "SMV" implémente un contrôle de type régime glissant (sliding mode) [47] dans le plan vertical. Ainsi, il reçoit la référence et la mesure en profondeur et tangage et produit la consigne à appliquer aux actionneurs verticaux.

Port	*Données produites*	*Type*	*unité*
0	vecteur état	cf. module NAV port 0	
1	valeurs Désirées	cf. module GSV port 0	
Port	*Evénements produits*	*Type*	*unité*
0	angle gouvernes avant	double	rad
1	angle gouvernes arriere	double	rad

RGM

Le module "RGM" est un module fixant le régime moteur à partir de la consigne reçue en paramètre.

Port	*Donnée consommée*	*Type*	*unité*
0	valeurs Désirées	cf. module GSV port 0	
Port	*donnée produite*	*Type*	*unité*
0	régime moteur	double	%

RGA

Le module "RGA" est un module qui permet de forcer les actionneurs à une consigne désirée, reçue en paramètre. Il est utilisé lors des phases de plongée.

Port	*Donnée reçue en paramètre*	*Type*	*unité*
0	valeurs actionneurs	struct { double delta_bow	*rad*
		double delta_stern	*rad*
		double delta_rudder	*rad*
		double delta_regime_moteur	%
		int use_mask }	
Port	*Données produites*	*Type*	*unité*
0	gouvernes avant	double	rad
1	gouvernes arrière	double	rad
2	gouvernes de cap	double	rad
3	régime moteur	double	%

10.3.4 Modules actionneurs

ACT

Le module "ACT" reçoit les consignes actionneurs et les applique après conversion.

Port	*Données consommées*	*Type*	*unité*
0	gouvernes avant	double	rad
1	gouvernes arrière	double	rad
2	gouvernes de cap	double	rad
3	régime moteur	double	rad

ATS

Le module "ATS" permet de rediriger les consignes actionneurs vers le simulateur, en phase de simulation.

Port	*Données consommées*	*Type*	*unité*
0	gouvernes avant	double	rad
0	gouvernes arrière	double	rad
0	gouvernes de cap	double	rad
0	régime moteur	double	rad

10.4 Module ordonnanceur

A chaque sous-objectif correspond l'exécution périodique d'un certain nombre de modules dans un ordre précis. Ces informations relatives à la correspondance entre un sous-objectif et un ensemble de modules sont contenues dans une base de données consultée par l'ordonnanceur (figure 10.2). A chaque réception de sous-objectif, l'ordonnanceur construit une sous-configuration. Une sous-configuration contient :

- la liste des tâches à exécuter ($\{A_1, ...A_m, T_1, ...T_n\}$),
- les paramètres temporels des tâches ($A_i =< r, R, P, C^-, C^+, L, C^-(t), C^+(t), L(t) >$ et $T_i =< r, R, P, C, C^-(t) >$),

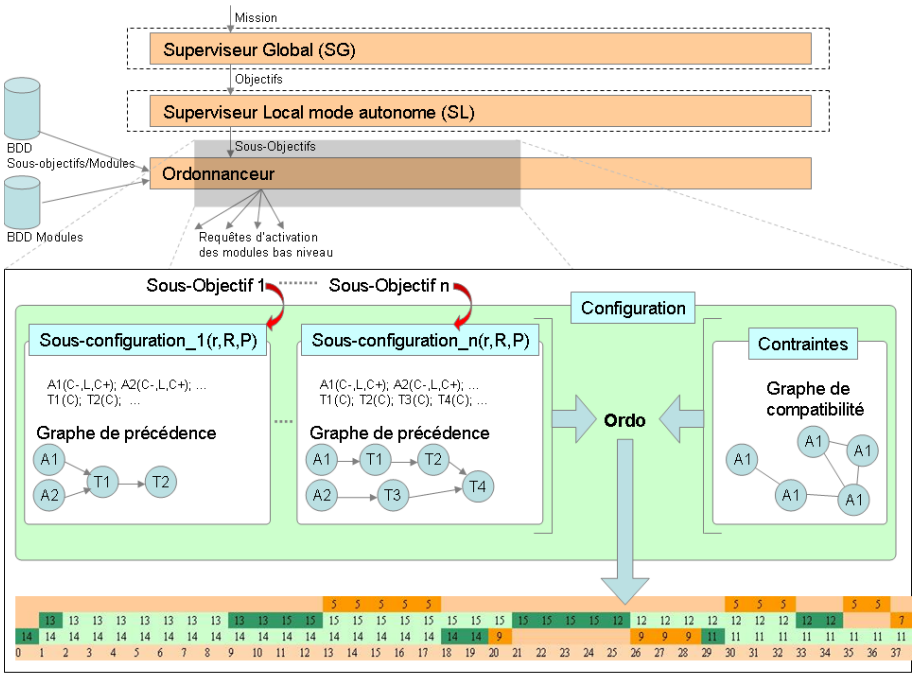


Figure 10.2 – Ordonnancement logico-temporel des modules

- le graphe de précédence entre les tâches (G_prec),
- la période de répétition de la sous-configuration (P),
- le délai critique d'exécution de la sous-configuration (R).

Les paramètres $C(t), C^-(t), C^+(t)$ sont appelés charge résiduelle de la tâche, c'est la charge restante à l'instant t. Par extension nous appelons $L(t)$ la latence résiduelle, elle correspond au temps restant avant la réponse supposée du capteur correspondant à la tâche.

L'ensemble des sous-configurations présents à un moment donné au sein de l'ordonnanceur forment une configuration. Dans notre implémentation, nous avons ajouté une base de donnée appelée "BDD Module" qui contient tous les paramètres temporels des tâches ainsi qu'un graphe de compatibilité.

En considérant l'ensemble des tâches de la configuration, leurs paramètres temporels, les graphes de précédence et le graphe de compatibilité, l'ordonnanceur est en mesure de déterminer qu'elle tâche doit s'exécuter à un moment donné.

Le module ordonnanceur (voir modèle RDP simplifié figure 10.3) fonctionne de la manière suivante : lorsqu'un sous-objectif est reçu, il crée la sous-configuration correspondante. L'algorithme d'ordonnancement est lancé sur la nouvelle configuration et produit un résultat correspondant à la date de début et la date de fin de la prochaine tâche à exécuter. Un timer est armé sur la date de début. Une fois que ce timer a déclenché, on lance la tâche (i.e. envoi d'une requête d'activation au module correspondant) et on arme le timer sur sa date de fin prévue. Si on reçoit un acquittement de cette tâche cela veut dire qu'elle s'est bien exécutée, on continue à ordonnancer. Si on reçoit un événement du timer en premier (i.e. avant l'acquittement de la tâche en cours) cela traduit un retard ou un blocage de la tâche.

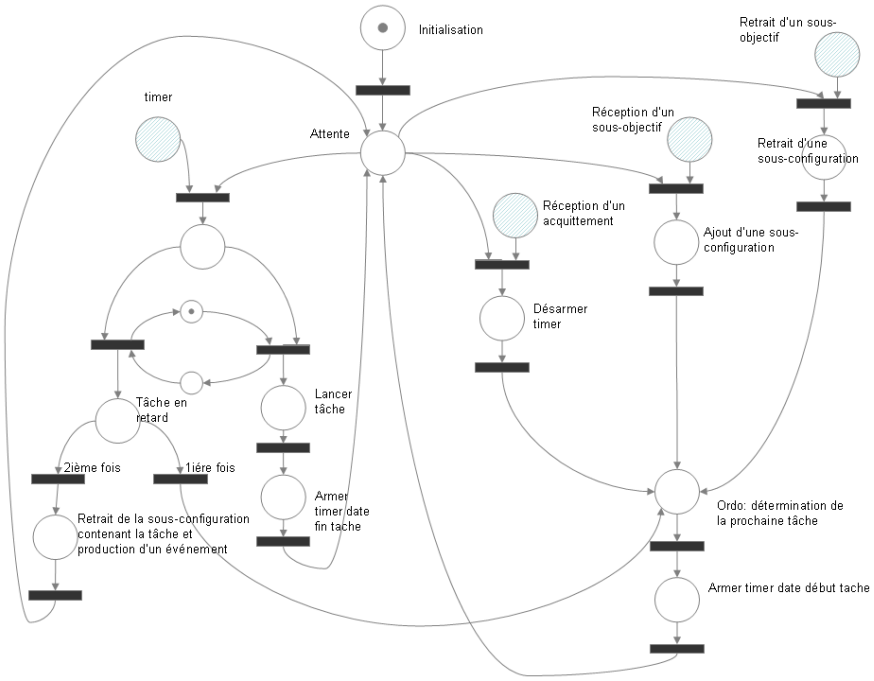


Figure 10.3 – Modèle RDP simplifié du module ordonnanceur

10.4.1 Algorithme d'ordonnancement

A partir d'une configuration de tâches données (e.g. figure 10.4), l'ordonnanceur sélectionne l'ensemble des tâches éligibles puis les classe par ordre de priorité. La tâche la plus prioritaire est élue.

Structure

L'algorithme d'ordonnancement (voir organigramme figure 10.5) divise le temps en pas. Pour chaque pas, un ensemble de calculs sont faits pour déterminer au final la date de début et la date de fin de la prochaine tâche à exécuter. Lorsqu'on parle d'exécution durant la description des étapes de l'algorithme, il faut entendre "simulation d'exécution". En effet l'algorithme simule l'exécution des tâches de la configuration à chaque pas de temps. C'est le résultat de cette simulation qui va fournir au module ordonnanceur les dates de début et de fin prévues pour la prochaine tâche.

Nous allons détailler davantage les différentes étapes de cet algorithme telles que numérotées sur la figure 10.5 :

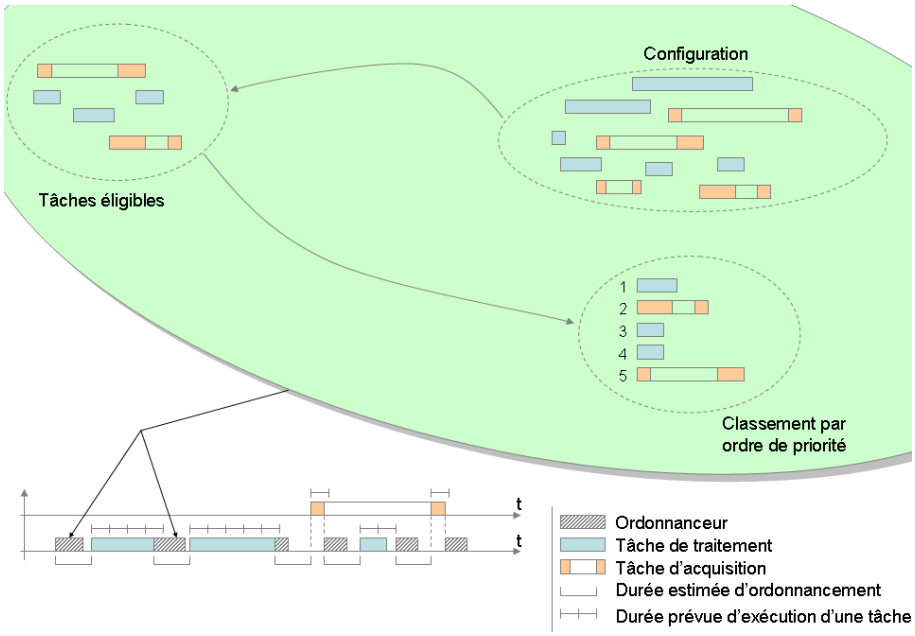


Figure 10.4 – Fonctionnement de l'ordonnanceur

Etape	*Description du traitement effectué dans cette étape*
1.	On considère un nouveau pas de temps. On réinitialise les charges et latences résiduelles des tâches ($C(t) = C$ ou $C^-(t) = C^-, L(t) = L, C^+(t) = C^+$) et on met à jour la date de réveil de la sous-configuration ($r_{S_c} = r_{S_c} + P_{S_c}$) dans le cas où on arrive dans une nouvelle période de la sous-configuration (i.e. si $r_{S_c} + P_{S_c} < t$).
2.	l'établissement de la liste des tâches éligibles, consiste à rechercher dans la configuration entière les tâches qui ne sont pas encore terminées ($C(t) > 0$ ou $C^+(t) > 0$ selon le type de tâche) et dont toutes les tâches qui les précèdent sont terminées.
3.	la priorité des tâches est la somme pondérée de l'échéance et du type de la tâche (cf. chapitre 7).
4.	on sélectionne la tâche la plus prioritaire de la liste des tâches éligibles. Si la liste est vide on va en 1.
5.	si la tâche la plus prioritaire est une tâche de traitement, elle peut être ajoutée directement à la liste (aller alors en 8). Si c'est une tâche d'acquisition, on doit vérifier qu'elle peut être lancée (compatibilité avec les autres tâches). En effet une autre tâche d'acquisition non compatible (et dont la zone L n'est pas encore terminée) peut se trouver dans la liste des tâches élues.
6.	en considérant le graphe de compatibilité, nous pouvons déterminer si la tâche d'acquisition considérée est compatible (reliée par un arc sur le graphe) avec les autres déjà élues. En cas d'incompatibilité on va en 13.
7.	dans le cas où elle est compatible, nous devons vérifier aussi que ses sous-tâches (C^+ et C^-) ne vont pas coïncider avec celles d'une des tâches acquisition déjà présentes dans la liste des tâches élues. Dans le cas d'incompatibilité on va en 13.
8.	On ajoute la tâche sélectionnée à la liste des tâches élues.

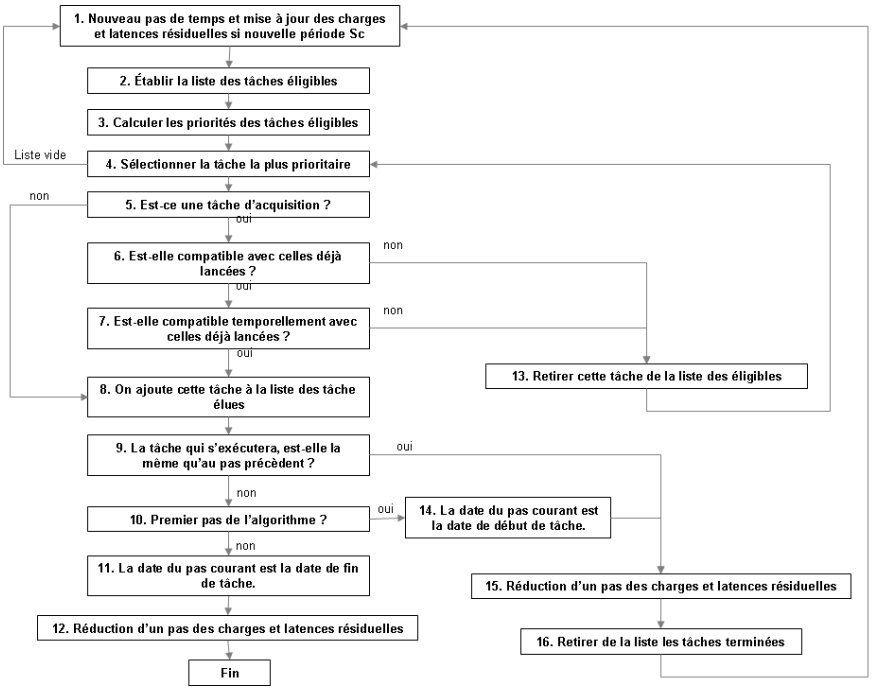


Figure 10.5 – Organigramme simplifié de l'algorithme de calcul d'ordonnancement

Etape	*Description du traitement effectué dans cette étape*
9.	ici on considère la liste des tâches élues. Si cette liste contient plusieurs tâches c'est qu'elles sont toutes sauf éventuellement une, des tâches d'acquisition dont la zone notée L est commencée mais pas encore terminée. l'éventuelle tâche restante est soit une tâche de traitement, soit une tâche d'acquisition dont l'exécution en est à la zone C^- ou C^+. C'est cette dernière alors qui demande à occuper l'unité de calcul. Si cette dernière s'était déjà exécutée au pas précédent alors on va en 15.
10.	s'il s'agit du premier pas de l'algorithme (i.e. premier pas de ce cycle d'ordonnancement) on va en 14.
11.	dans le cas où cette tâche ne s'est pas exécutée au pas précédent et que nous ne nous trouvons pas au premier pas de l'algorithme, cela signifie qu'une tâche venait de se terminer ou qu'elle a perdu le processeur au pas précédent. On sauvegarde alors la date du pas de temps courant car il correspond à la date d'arrêt de la tâche du pas précédent. Arrivé à ce point on a déterminé la date de début et celle de fin de la tâche à exécuter.
12.	on réduit les charges et latences résiduelles des tâches élues. Cela revient à simuler une exécution d'un pas de ces tâches. Fin du cycle d'ordonnancement.
13.	lorsqu'une tâche d'acquisition n'est pas compatible avec celles déjà élues nous devons l'exclure de la liste des tâches éligibles. En effet, c'est le seul choix qui se présente puisque les tâches d'acquisition ne sont pas préemptibles.
14.	la date du pas courant est la date de début de la tâche considérée.
15.	on réduit les charges et latences résiduelles des tâches élues. Cela revient à simuler une exécution d'un pas de ces tâches.
16.	les tâches élues s'étant toutes exécutées d'un pas à cette étape, on retire de la liste des élues celles qui viennent de se terminer (i.e : $C(t) = 0$ ou $C^+(t) = 0$). On recommence alors 1.

Tolérance aux incertitudes paramétriques

Comme nous l'avons évoqué au chapitre 7, les données (paramètres temporels des tâches) sur lesquelles se base l'algorithme d'ordonnancement sont des estimations qui s'avèrent inadaptées aux valeurs réelles. L'algorithme doit adapter les estimations des paramètres des tâches en fonction des valeurs observées.

Pour calculer la durée réelle d'exécution des modules, le module ordonnanceur calcule le temps séparant l'envoi de sa requête d'activation et la réception d'un acquittement d'exécution.

Nous avons aussi besoin de connaître une estimation de la durée de l'ordonnancement puisqu'elle est prise en compte pour déterminer la date de début de l'échéancier (figure 10.4). De la même manière que pour le calcul de l'estimation des paramètres des tâches, l'estimation de la durée de l'ordonnancement va se baser sur les durées observées. Pour cela le module ordonnanceur va simplement calculer la durée de la fonction d'ordonnancement.

Pour tous ces paramètres, nous utiliserons la formule définie au chapitre 7 :

$$val_{estime} = moyenne + 1.5\sigma_n \tag{10.1}$$

avec

$$\sigma_n = \frac{1}{\sqrt{n}}\sqrt{\sum_{i=1}^{n}(x_i - \frac{1}{n}\sum_{i=1}^{n}x_i)^2} \tag{10.2}$$

σ_n est l'écart type de n valeurs observées. Nous établirons ces calculs sur un horizon glissant de 10 échantillons. Pour simplifier le calcul de l'écart type, nous transformons l'équation 10.2 :

$$\sigma_n = \sqrt{\frac{\sum_{i=1}^{n}x_i^2}{n} - (\frac{\sum_{i=1}^{n}x_i}{n})^2} \tag{10.3}$$

10.4.2 Flux de donnée/contrôle autour de l'ordonnanceur

Après initialisation, le module ordonnanceur est en attente sur sa boîte aux lettres requête. Il est sensible à quatre types de messages :

- **Ajout d'un nouveau sous-objectif**. Ce message lui est envoyé par un superviseur local sur son port de paramétrage n°0. Le message est une chaîne de caractères de la forme "nomSousObjectif($parm_1, param_2, ..., param_N$)". Le nom du sous-objectif est extrait de cette chaîne de caractères ce qui va permettre de le rechercher dans la base de données sous-objectif/modules. Une fois identifié, on construit au sein de la configuration de tâches courantes, une nouvelle sous-configuration composée des tâches décrites dans la base de données ainsi que d'autres paramètres tels que le graphe de précédence. De la chaîne de caractères reçue, sont également extraits les paramètres "$param_i$". Ces paramètres seront envoyés aux ports de paramétrage des modules. La correspondance entre les paramètres et leurs ports et modules de destination sont présents dans la base de données (voir 10.6).

- **Retrait d'un sous-objectif**. Ce message est aussi issu d'un superviseur local à destination du port de paramétrage n°1 du module ordonnanceur et il est de la forme "nomSousObjectif". Grâce à ce nom et en consultant la base de données l'ordonnanceur détermine la sous-configuration à retirer.
- **Message Timer**. Lorsque le timer armé par le module ordonnanceur arrive à échéance, il poste un message au port n°2. Ce message contient la date surveillée par le timer.
- **Requête d'Acquittement**. Lorsque le module ordonnanceur envoie une requête d'activation à un module, elle est accompagnée d'un identifiant (numéro de la tâche correspondante). Cet identifiant est retourné à l'ordonnanceur dans la requête d'acquittement et permet de connaître le module qui a terminé son exécution.

Deux types de messages sont produits par le module ordonnanceur :

- **Envoi des paramètres au module**. Certains modules requièrent des paramètres (e.g. consigne pour une loi de commande) pour pouvoir s'exécuter. Ces paramètres sont définis par l'utilisateur lors de la mise en place du vocabulaire (voir 10.6). Ils peuvent provenir soit de la base de données soit être extraits de la chaîne de caractères, représentant un sous-objectif, reçue de la part du superviseur local. Dans tous les cas l'ensemble des paramètres à envoyer aux modules est propre au sous-objectif dans lequel ils sont engagés. Ces paramètres sont envoyés une seule fois lors de la réception du sous-objectif.
- **Envoi des requêtes de START**. L'algorithme d'ordonnancement fournit une date de début et une date de fin pour l'exécution d'un module donné. Lorsque la date de début d'exécution du module est arrivée, le module ordonnanceur envoie une requête START au module et augmente sa priorité.

Note : L'ordonnanceur joue sur deux niveaux de priorité, plus basses que la sienne :

- priorité basse pour tous les modules suspendus,
- priorité moyenne pour le module actif.

10.5 Modules haut-niveau

Les modules haut-niveau traitent de l'aspect stratégique de la mission. Ils prennent des décisions en appliquant un ensemble de règles, prédéfinies, sur l'état du système. L'état du système est traduit par des états abstraits et des événements produits par les modules du bas niveau auxquels les différents superviseurs s'abonnent. Leur comportement est complètement asynchrone et événementiel.

Un point essentiel concerne leur articulation avec l'ordonnanceur. En effet comme nous l'avons évoqué jusqu'ici, l'ordonnanceur contrôle et surveille l'activité des modules bas niveau de l'architecture. Il base son ordonnancement sur sa connaissance des propriétés temporelles de l'exécution des modules et sur l'hypothèse qu'aucun module ne puisse s'exécuter sans son autorisation.

Cependant, cet ordonnanceur n'est pas en mesure de gérer aussi l'exécution des modules implémentant les superviseurs. D'abord, ces modules sont d'un niveau hiérarchique supérieur, pour pouvoir demander à l'ordonnanceur de les activer, ils leur faut déjà être en activité. De plus l'exécution des superviseurs est basée sur les événements par nature imprévisibles donc ne pouvant être pris en charge par un ordonnancement.

La solution retenue est d'attribuer aux superviseurs une priorité supérieure à celle de l'ordonnanceur. Ils seront réveillés par les événements et préempteront tous les modules d'un niveau inférieur, y compris l'ordonnanceur. Cette solution ne peut être valide au regard des applications (temps-réel) désirées que si l'on fait l'hypothèse que :

- le nombre d'événements à traiter est faible,
- la durée de traitement des événements par les superviseurs est petite comparée à la dynamique du système,
- l'ordonnanceur est capable de palier aux retards engendrés.

Les superviseurs décident de ce qui est à faire dans le contexte courant, l'ordonnanceur exécute ce qui est à faire dans le contexte courant. Donc pour d'évidentes considérations de réactivité, la priorité est donnée à la décision de ce qui est à faire au regard des événements qui se produisent. L'ordonnanceur exécutera alors la décision des superviseurs...i.e. potentiellement arrêter le sous-objectif en cours pour en lancer un autre. Quant à la fréquence d'interruption de l'ordonnanceur au profit des superviseurs, elle dépend de la répartition de la prise en compte des phénoménes significatifs au sein de l'architecture : plus le bas niveau gère des événements, moins les superviseurs seront sollicités. Mais cette répartition doit être pertinente, c'est à dire que l'événement doit être adressé à l'entité capable de le gérer.

Le traitement d'un événement par un superviseur revient à l'évaluation des équations logiques qui l'impliquent, ce qui représente un temps de traitement très faible comparé aux calculs effectués par les lois de commande.

Enfin l'activation d'un superviseur aura pour effet de préempter un module en cours d'exécution puis de lui rendre la main à la fin du traitement. Du point de vue de l'ordonnanceur, cela sera perçu comme un retard d'exécution du module préempté...si l'activation de la supervision ne donne pas comme résultat un changement de sous-objectif. Nous avons vu précédemment que l'ordonnanceur est capable de s'adapter aux variations de durées d'exécution des modules.

10.5.1 Superviseur Global

Le superviseur global est un module qui après initialisation se met en attente d'une mission à effectuer. Cette mission est reçue sous la forme d'une chaîne caractères contenant le nom de la mission et ses paramètres éventuels : "mission($param_1, ..., param_N$)". Lors de sa réception sur son port 0, le module superviseur global consulte la base de données pour déterminer les objectifs à lancer pour accomplir cette mission ainsi que la correspondance entre les paramètres reçus ($param_i$) et ceux à donner aux objectifs. Ce module s'abonne aux événements nécessaires à l'évaluation des équations logiques de lancement et d'arrêt des objectifs. Lorsqu'un objectif doit être lancé, il envoie une requête au superviseur local contenant le nom de l'objectif ainsi que ses paramètres.

Nous avons évoqué au chapitre 6 que le superviseur global devait gérer les ressources et leurs modes à un niveau macroscopique (macro-ressource) et choisir en fonction des tâches à faire le superviseur local associé (i.e. dédié à ce couple ressource-mode). Dans le cas du véhicule sous-marin que nous étudions, nous identifions qu'une seule macro-ressource, le véhicule lui-même et nous pouvons envisager trois modes : le mode autonome, le mode téléopération (en surface) et le mode coopération. A ce stade des développements, seul le superviseur local du mode autonome a été implémenté, le mode téléopération de surface

étant parallélement étudié.

10.5.2 Superviseur local

Le superviseur local du mode autonome développé est un module en attente sur son port requête paramétrage n°0 d'un message contenant un objectif. Ce message est une chaîne de caractère contenant le nom de l'objectif à lancer avec une liste de paramètres : "objectif($param1, ..., paramN$)". En consultant la base de données objectif/sous-objectif, le superviseur local détermine les sous-objectifs à lancer pour réaliser l'objectif ainsi que les équations logiques à évaluer pour connaître les moments où chacun de ces sous-objectifs doit être déclenché ou arrêté. Il effectue dans ce sens l'ensemble des abonnements aux événements impliqués dans les différentes équations logiques.

Pour lancer un sous-objectif, il en transmet le nom ainsi que les paramètres au module ordonnanceur.

10.6 Bases de données

10.6.1 Implémentation

Comme nous l'avons vu le fonctionnement de notre architecture est basée sur un vocabulaire. Ce vocabulaire permet de réifier les capacités de l'engin ainsi que les intentions qu'on peut lui demander. Il donne alors une identité à notre architecture puisque deux architectures développées pour deux systèmes robotiques différents vont trouver leur principale différence dans le vocabulaire (et les modules associés).

Ce vocabulaire se trouve dans des bases de données consultées par les différentes entités de l'architecture. Nous avons implémenté les différentes bases de données de l'architecture sous forme de fichiers. Une base de donnée consultable et modifiable dynamiquement permettrait d'enrichir dynamiquement le vocabulaire (i.e. en cours de fonctionnement). Cela n'a pas d'intérêt dans notre cas où les missions sont effectuées sans lien de communication avec l'opérateur. On pourra seulement intervenir sur le vocabulaire hors mission.

Les fichiers de vocabulaire sont lus par les modules concernés qui en font une représentation en mémoire, représentation sur laquelle ils se basent pour traiter les requêtes qu'ils reçoivent. Afin d'assurer l'unicité dans la syntaxe écrite par l'utilisateur et celle d'interprétation des fichiers lus, nous avons défini un format pour chaque type de déclaration. Ces formats sont basés sur des motifs que doit reconnaître le module lecteur de ce type de fichier. Par exemple "plonger(-100)" et "plonger (- 1.10e2)" doivent signifier la même chose. Par contre "plo_nger(-10.5)" ne sera pas du tout reconnu.

Beaucoup d'outils ont été développés pour effectuer de la reconnaissance d'expression. Ils se basent tous sur un langage de description plus ou moins commun des motifs, appelé "expressions régulières". Voici par exemple l'expression régulière que nous utilisons pour décrire le nom des modules : `"[a-zA-Z][a-zA-Z1-9]{2}"`. Cela signifie que le premier caractère du nom d'un module est une lettre de l'alphabet (entre a et z ou entre A et Z), les caractères suivants seront chacun soit une lettre de l'alphabet soit un chiffre.

A partir d'une expression régulière les outils sont capables de trouver et d'extraire d'une ligne les motifs auxquels elle correspond. Outre l'avantage de disposer d'un ensemble d'outils performants dans la recherche de motifs, cette façon de procéder nous permet en

outre de pouvoir modifier très facilement la syntaxe de notre vocabulaire sans avoir à intervenir au sein d'un algorithme. Si nous utilisons un système d'exploitation temps réel qui nous impose par exemple d'utiliser des noms ne contenant pas de chiffre pour nommer les tâches, il suffira de modifier l'expression régulière correspondante.

Pour notre cas nous utilisons la bibliothèque C "regex" pour rechercher les expressions régulières. Par contre nous construisons et testons les expressions avec le langage "perl". Ce langage permet une construction incrémentale des expressions ce qui permet de modifier facilement les longues expressions.

10.6.2 Mission/Objectif

La mission suivante a été testée au lac :

Mission lac du Salagou

```
missionSalagou()
{
   [tps = 1s]     watchdog()     [duree=15min];
   [tps = 2s]     chemin()     [duree=4min];
   // En cas de problème d'ordonnancement, on remonte à la surface
   [ORD:0 or ORD:1]     surface()     [PRF:0];
}
```

10.6.3 Objectif/Sous-objectif

Voici les objectifs utilisés par la mission tels que décrits dans le fichier Objectif/Sous-objectif :

extrait fichier Objectif-Sous-objectifs

```
watchdog()
{
   []     watchdog()     [];
}
chemin()
{
    1: [tps=5s]     setpointPIG(1.8, 0, 300, 0, 1.8)     [duree=50s];
    2: [endof(1)]     setpointPIG(1.8, 0, 300, 3, 1.8)     [duree=50s];
    3: [endof(2)]     setpointPIG(1.8, 0, 300, 0, 1.8)     [duree=50s];
}
surface()
{
   []     setpointPIG(1.8, 0, 300, 0, 0) [PRF:0];
}
```

10.6.4 Sous-objectif/Module

Plusieurs sous-objectifs ont été développés pour les tests sur Taipan 300, nous ne présenterons ici que les sous-objectifs qui sont impliqués dans la mission effectuée en lac :

extrait fichier Sous-objectifs/Modules

```
watchdog()
{
    PERIODE = 1s;
    DELAI_CRITIQUE = 1s;
    DIO();
}
setpointPIG(double u, double theta, double Cap_Des, double Prof_Des, ...
                                                        ... double Mot_Vel)
{
   // parametres du sous objectif
    PERIODE = 100 ms;
    DELAI_CRITIQUE = 100 ms;
   // modules mis en oeuvre et parametres respectifs
    GSV({u,theta,Cap_Des,Prof_Des,Mot_Vel});
    MTI();
    PRF();
    NAV();
    PIG({20,10,0,100,10,0,10,10,0,1,1,0.1,1,0.5,1});
    RGM();
    ACT();
   // graph de precedence
    GSV -> PIG;
    GSV -> RGM;
    MTI -> NAV;
    PRF -> NAV;
    NAV -> PIG;
    PIG -> ACT;
    RGM -> ACT;
   // Liens de communication avec periodicite (*periodicite)
    GSV:0 -> PIG:1 *1;
    GSV:0 -> RGM:0 *1;
    MTI:0 -> NAV:2 *1;
    MTI:1 -> NAV:3 *1;
    PRF:0 -> NAV:1 *1;
    NAV:0 -> PIG:0 *1;
    PIG:0 -> ACT:0 *1;
    PIG:1 -> ACT:1 *1;
    PIG:2 -> ACT:2 *1;
    RGM:0 -> ACT:3 *1;
}
```

10.6.5 Module

Le fichier des modules contient la définition des modules ainsi que le graphe de compatibilité :

fichier Modules

```
GSV(5ms)({double, double, double, double, double});
WPT(5ms)({double,double, double, double, double},{double, double, double});
NAV(5ms);
PIH(5ms);
PIG(5ms)({double,double,double, double, double, double, double, double,...
            ... double, double, double, double, double, double, double});
RGM(5ms)();
SMH(5ms);
SMV(5ms)();
ACT(30ms);
DIO(5ms);
PRF(5ms);
GPS(5ms);
MTI(10ms);
RGA(5ms)({double, double, double, double});
SAV(10ms);
CTD(10ms);
SON(10ms);
// Modules pour le simulateur
STC(10ms);
ATS(10ms);
```

Résumé du chapitre 10 :

- Pour faciliter les développements, nous avons mis en place un environnement de travail centré autour d'un serveur contenant l'architecture logicielle et un ensemble d'outils informatiques.
- 19 modules ont été développés pour gérer les différents capteurs et traitements de Taipan 300. 3 modules implémentent respectivement, un superviseur global, un superviseur local et un ordonnanceur.
- Les bases de données de vocabulaire ont été implémentées sous forme de fichiers.
- Nous avons défini le vocabulaire (mission, objectif, sous-objectif) nécessaire à l'exécution d'une mission simple.

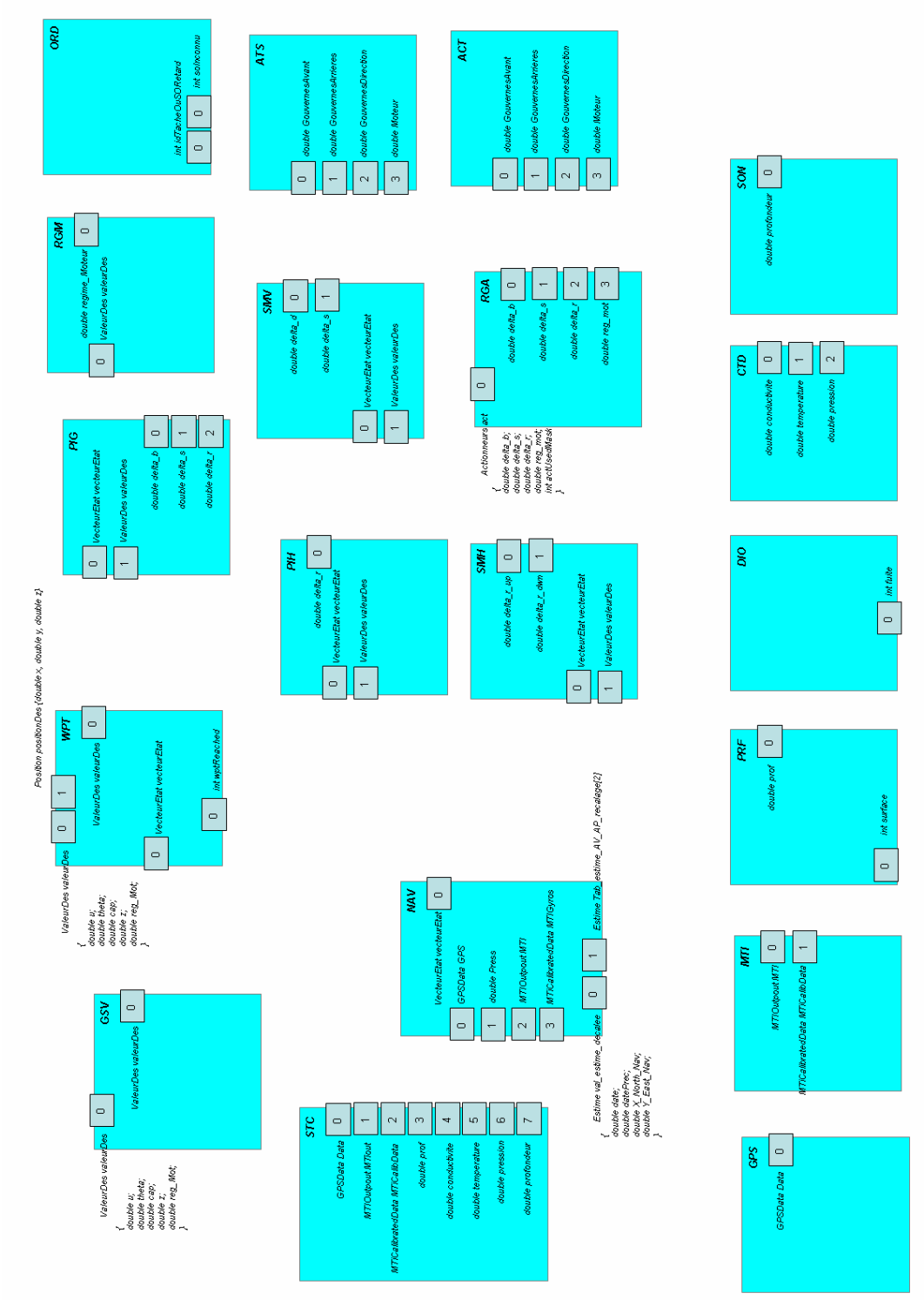


Figure 10.6 – Représentation des modules développés pour taipan 300

Chapitre 11

Expérimentations et résultats

11.1 Protocole expérimental

Afin de valider expérimentalement l'architecture logicielle proposée, nous avons choisi de l'implémenter sur le robot *Taipan300* présenté précédemment. Nous avons mené nos expérimentations dans le Lac du Salagou situé au sud de Lodève, à 50 km à l'ouest de Montpellier (France). Ce choix est en partie dû à sa proximité par rapport au laboratoire, mais également à l'assurance de trouver une météo clémente. En effet, même par vent soutenu, la houle reste négligeable, et ne nous gêne en rien lors des manipulations. D'autre part, la configuration des lieux autorise de mettre l'engin à l'eau depuis la rive, permettant d'éviter ainsi l'utilisation d'un bateau à moteur, par ailleurs interdit sur le lac. La flottabilité de l'engin est suffisante pour lui permettre de rester en surface, même si l'eau du lac est douce.
Le matériel mis en œuvre pour effectuer ces expérimentations est assez limité. En effet, outre l'engin lui-même, nous transportons un PC portable durci (doté d'un écran tactile et d'une batterie d'une large capacité qui autorise une autonomie de près de 6H), une radio et sa batterie nous permettant de communiquer avec la torpille à longue distance (environ 3km), et enfin de l'outillage nécessaire au montage/démontage de l'engin. Un GPS nous permet de nous positionner par rapport à la torpille sur une carte construite localement en parcourant le rivage.
La logistique et la mise œuvre de *Taipan300* sont relativement aisées. Le faible encombrement et le poids limité de cet engin, nous permettent de le manipuler aisément (notamment lors des phases de mise à l'eau ou de récupération de l'engin), mais également de le transporter facilement dans le coffre d'une voiture. Nous avons donc déployé notre architecture logicielle sur *Taipan300* en effectuant une simple copie de celle développée sur *Perle*. La prise de contrôle de l'engin et la programmation des missions se fait très facilement grâce à l'utilisation d'un réseau WIFI AD-Hoc. Il nous suffit donc de nous connecter en *ssh*, pour compiler/lancer les modules de l'architecture, ou ajouter une mission dans le vocabulaire.

Le lancement d'une mission s'effectue en communiquant au superviseur global le nom de la mission à exécuter. L'engin est mis à l'eau depuis la rive, et l'ordre de départ est

Figure 11.1 – Expérimentation de l'architecture logicielle sur l'AUV Taipan 300

donné. Par mesure de sécurité nous réalisons les tests en prenant soin de relier la queue de la torpille à une ficelle dont l'autre extrémité est fixée à terre. En effet, comme nous l'avons mentionné précédemment, nous de disposons pas de bateau pour aller récupérer la torpille en cas d'ennuis techniques. D'autre part, cela nous permet de limiter les risques de collision lorsqu'elle est en surface (possible présence de baigneurs, embarcations en tout genre, retour berge etc...). Pour tester l'architecture logicielle, nous avons souhaité programmer une mission relativement simple en s'attachant à tracer (logger) toute l'activité du contrôleur. La mission consistait donc en une succession de 3 lignes droites orientées au cap désiré 300°, à une vitesse de $1.8 m.s^{-1}$ et pour des profondeurs désirées de 0m, 3m et 0m, le dernier palier s'effectuant à propulsion nulle. La loi de commande implémentée (au sein du module PIG) est du type "PID guidé", loi de commande linéaire asservissant l'engin à suivre un guidage non-linéaire. La torpille effectue donc une première ligne droite en surface, puis plonge à 3m de fond et enfin revient en surface grâce à sa flottaison positive. L'ensemble des actionneurs est donc sollicité, ainsi qu'une partie des capteurs (capteur de pression, centrale inertielle, GPS).

11.2 Résultats relatifs à la mission

Chaque module enregistre les données qu'il manipule selon un format commun. A chaque activation, il sauvegarde les valeurs des données produites sur chacun de ses ports avec les dates de production. Nous avons conçu un petit applicatif dédié à exploiter ce type de fichier : un analyseur de log permettant de visualiser toutes les traces enregistrées. Pour évaluer qualitativement la réussite de la mission, nous utilisons les données de position reconstruites à partir des données de la centrale d'attitude et du capteur de pression

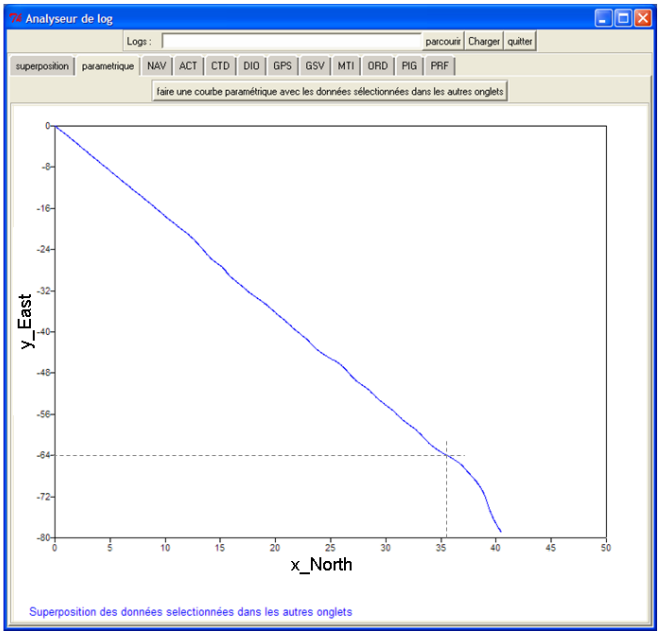


Figure 11.2 – Trajectoire de l'AUV calculée à partir de l'estimation de déplacement

embarqués dans le véhicule. Cette reconstruction est assurée par le module NAV qui produit la position estimée en direction du Nord (x_{North}) et la position estimée en direction de l'Est (y_{East}). On en déduit facilement la cap suivi par la torpille :

$$\alpha_{cap} = arctan(\frac{-64}{35.5}) = -1.06 = -60.98^{\circ} = 299.01^{\circ} \tag{11.1}$$

Nous obtenons une trajectoire rectiligne au cap d'environ 300° ce qui a été la consigne, comme nous l'indique la figure 11.3.

En ce qui concerne la tenue de profondeur, les figures 11.4 et 11.5 montrent une stabilisation de l'engin autour de la consigne désirée à tangage quasi-nul. Il est à noter que le dernier plateau s'effectue à propultion nulle, l'engin regagnant la surface grâce à sa flottabilité positive, ce qui explique la lente remontée en fin de mission. L'analyse de l'évolution de la gouverne avant montre la réaction des actionneurs à la consigne, la saturation positive en fin de mission s'expliquant par une compensation du tangage positif que la remontée statique fait prendre à l'engin.

Ces résultats sont suffisamment significatifs pour valider qualitativement le comportement de l'architecture implémentée. Cependant, il est évident que la loi de commande nécessite une analyse plus poussée, notamment en ce qui concerne l'évolution de l'angle de roulis au cours de la mission (figure 11.6). En effet la prise de roulis induit un couplage indésirable entre les plans horizontaux et verticaux et engendre des défauts sur le suivi des consignes.

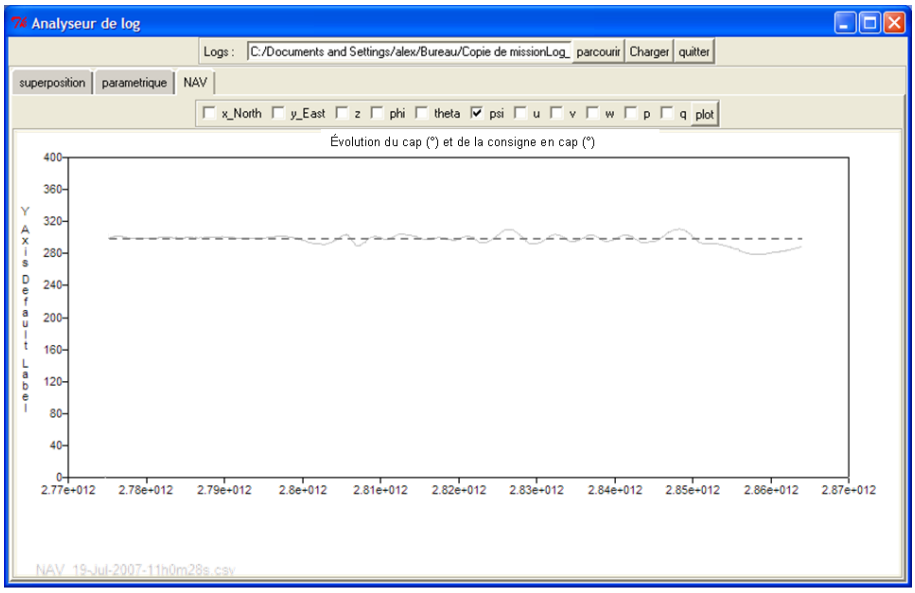


Figure 11.3 – Evolution du cap (degrés) et consigne en cap (degrés)

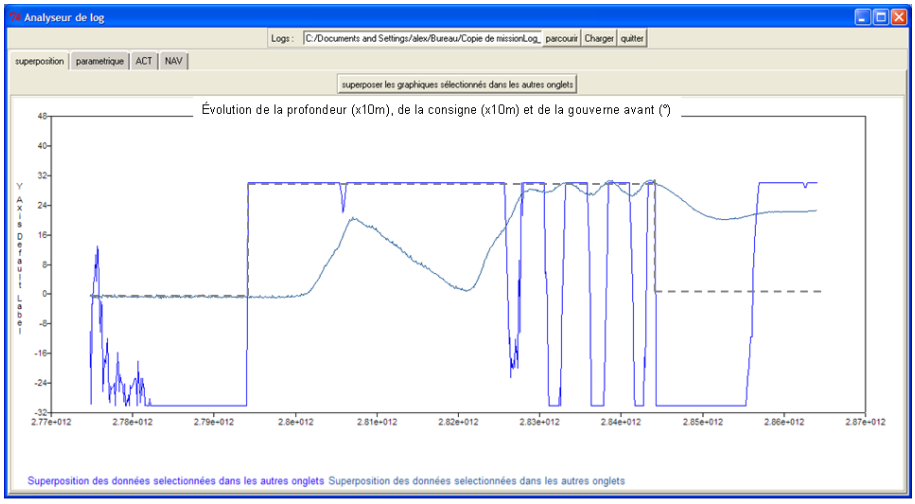


Figure 11.4 – Evolution de la profondeur (x10m), de la consigne (x10m) et de la gouverne avant (degrés)

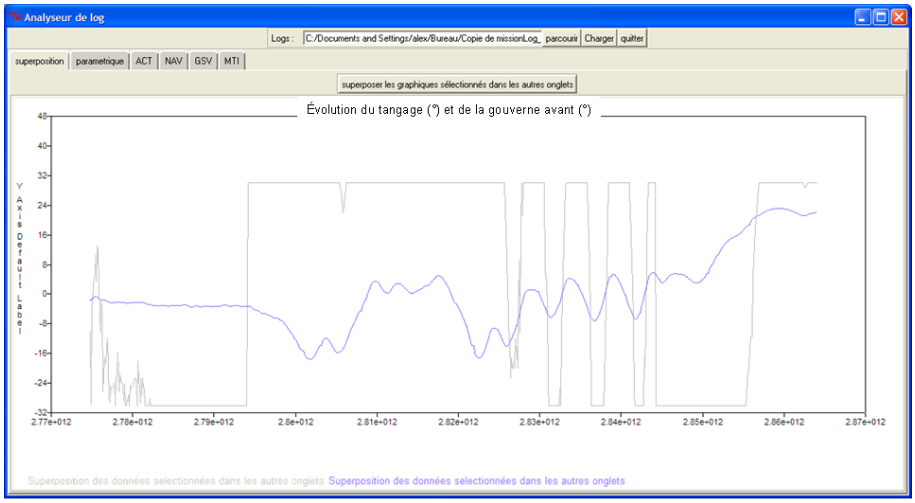


Figure 11.5 – Evolution du tangage (degrés) et de la gouverne avant (degrés)

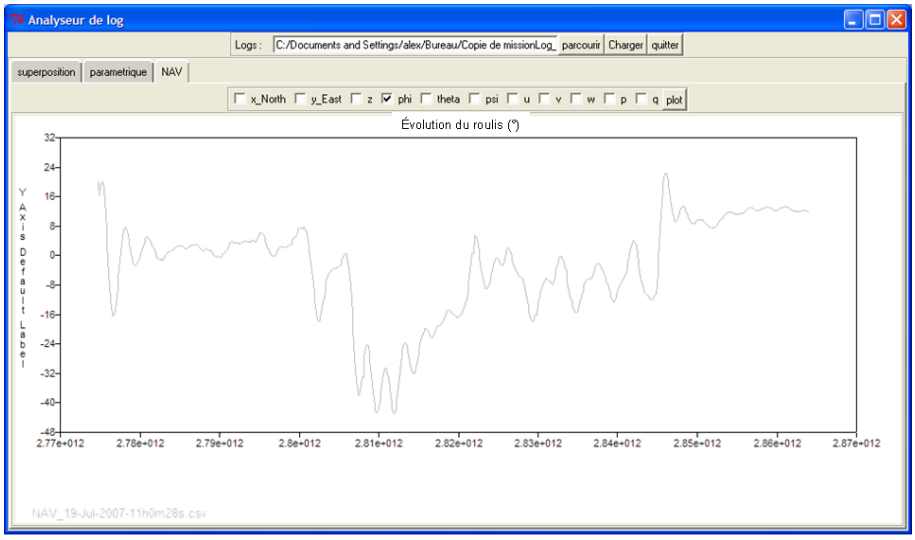


Figure 11.6 – Evolution du roulis (degrés)

11.3 Résultats relatifs à l'architecture logicielle

11.3.1 L'ordre d'enchaînement des modules et des sous-objectifs

L'ordonnanceur sauvegarde les données relatives aux dates :
- d'activation des modules,
- de réception des acquittements d'arrêt,
- de début de l'ordonnancement,
- de fin de l'ordonnancement,
- de réception d'un sous-objectif,
- de changement de période des sous-configurations correspondantes,
- d'arrêt des sous-objectifs.

Nous exploitons ces données pour pouvoir vérifier le respect de l'ordre d'enchaînement des modules ainsi que les durées des modules.

Pour faciliter l'interprétation des fichiers de log de l'ordonnanceur, nous représentons l'activité des modules par une fonction $f_{activite\,modules}$ telle que :

$$f_{activite\,modules}(t) = \begin{cases} 1 & \text{si ORD est actif à t} \\ 2 & \text{si MTI est actif à t} \\ 3 & \text{si PRF est actif à t} \\ 4 & \text{si NAV est actif à t} \\ 5 & \text{si GSV est actif à t} \\ 6 & \text{si RGM est actif à t} \\ 7 & \text{si PIG est actif à t} \\ 8 & \text{si ACT est actif à t} \\ 9 & \text{si DIO est actif à t} \end{cases}$$

La hauteur du créneau dans le graphe obtenu distinguera le module qui est en activité à un moment donné. De manière analogue nous représentons les changements de période des sous-configurations :

$$f_{activit\,sous-config}(t) = \begin{cases} -1 & \text{si } t = r_{watchdog()} + kP_{watchdog()} \text{ avec } k \in \mathbb{N} \\ -2 & \text{si } t = r_{setpointPIG()} + kP_{setpointPIG()} \text{ avec } k \in \mathbb{N} \end{cases}$$

L'analyse de la figure 11.7.(a) indique une zone dense en début de mission, qui correspond à la mission effective. La zone suivante correspond à l'activité unique du module DIO (utilisé par le sous-objectif watchdog), alors que la torpille est revenue en surface et attend sa récupération.

L'analyse de l'activité des modules lors de la mission effective (cf. figure 11.7.(b)) indique l'enchaînement attendu des différents modules impliqués.

La figure 11.7.(c) illustre l'activité de ces modules sur un horizon de deux périodes (200 ms). On constate l'enchaînement attendu respectant les contraintes de précédence telle qu'exprimées dans le fichier de définition du sous-objectif (voir section 10.6.4). On peut remarquer en outre que les modules impliqués dans le sous-objectif setpointPIG s'exécutent au sein d'une même période (cf. figure 11.7.(d)).

On peut observer des temps d'inactivité après l'exécution de chaque module. Ces temps morts correspondent à la différence entre la durée estimée de chaque module et la durée réelle. En effet lorsque l'ordonnanceur active un module, il lui octroie un délai

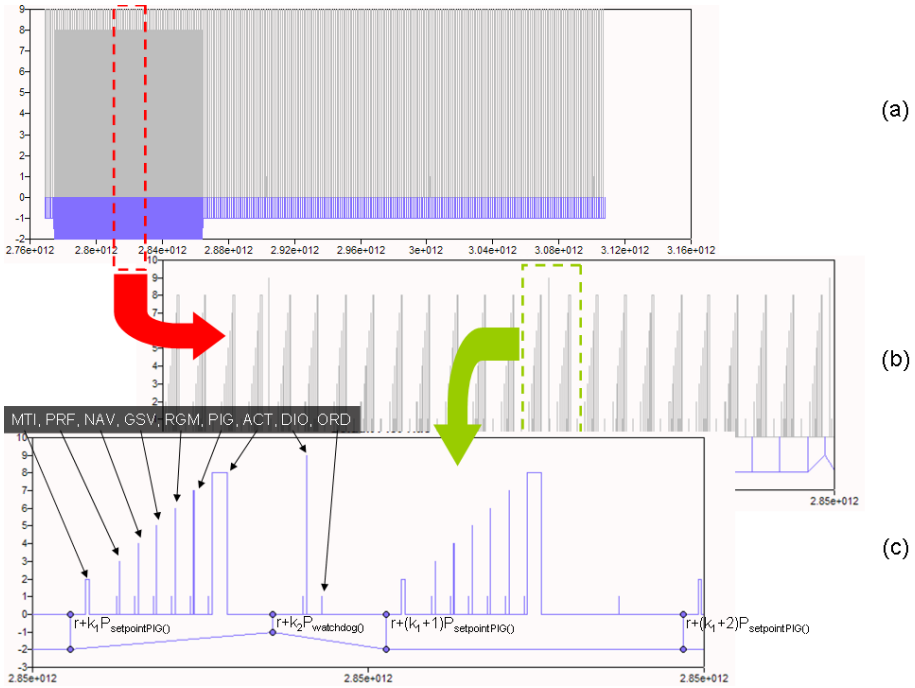


Figure 11.7 – Activité des modules et des sous-objectifs

d'exécution, une fois ce délai expiré, l'ordonnanceur reprend la main. Quand le module répond très en deçà de ce délai, on observe des temps d'inactivité.

11.3.2 Analyse des communications intermodules

La figure 11.8 compare la consigne de la gouverne de cap produite par le module PIG et la donnée appliquée par ACT à la gouverne de cap. On constate tout d'abord que l'application à la gouverne de cap de ACT correspond bien à la consigne produite par PIG. On constate comme attendu que le module ACT sature les consignes reçues en fonction des valeurs limites des servomoteurs.

De plus, les figures 11.8 et 11.9 indiquent clairement que la communication entre les modules PIG et ACT se déroule normalement. En effet, de la figure 11.8, il est possible d'extraire la différence de temps entre la production et la consommation d'une même variable, et de la figure 11.9, il est possible de mesurer le temps séparant l'activation de ces deux modules : ces temps attestent que la communication se déroule bien dans le cadre de la période (cycle de la sous-configuration).

11.3.3 Analyse temporelle de l'activité des modules et sous-objectifs

La figure 11.10 représente l'évolution des durées d'exécution de deux modules (MTI et NAV) pris pour exemple. On voit que pour les deux modules, les durées d'exécution fluctuent autour d'une valeur moyenne avec parfois de grands dépassements.

On constate que la durée moyenne d'exécution du module MTI est inférieure à 2 ms et celle du module NAV est inférieure à 1 ms. Ces valeurs observées sont inférieures aux valeurs données en estimation à l'ordonnanceur par l'intermédiaire du fichier de définition

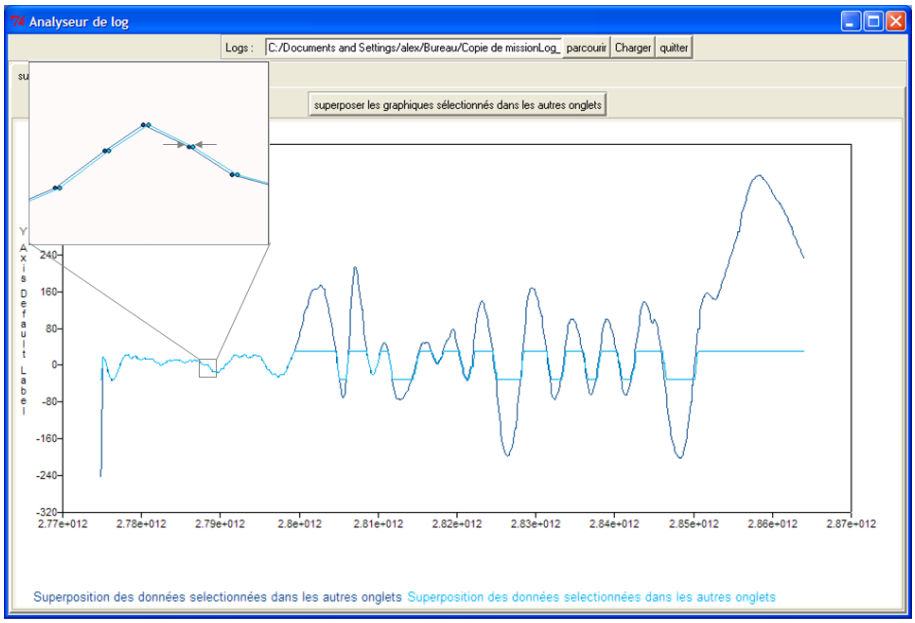


Figure 11.8 – Evolution de la consigne en cap produite par le module PIG, et évolution de la consigne envoyée à la gouverne de cap par le module ACT

des modules (voir section 10.6.5). Cela explique les temps d'inactivité observés à la figure 11.7.(d) (voir section 11.3.1)

L'option d'adaptation dynamique de l'estimation des durées d'exécution des modules (voir section 10.4.1 et 7) n'a pas été activée pour les tests menés au lac. Elle aurait permi de faire diminuer au cours du temps la durée des temps d'inactivité observés.

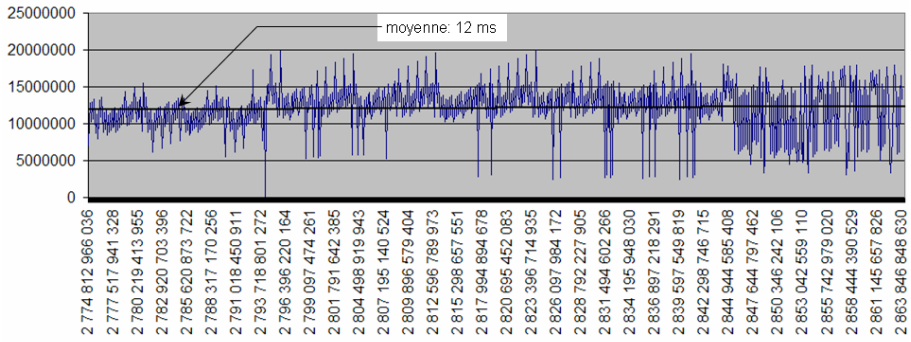


Figure 11.9 – Différence des dates de log des modules PIG et ACT (ns)

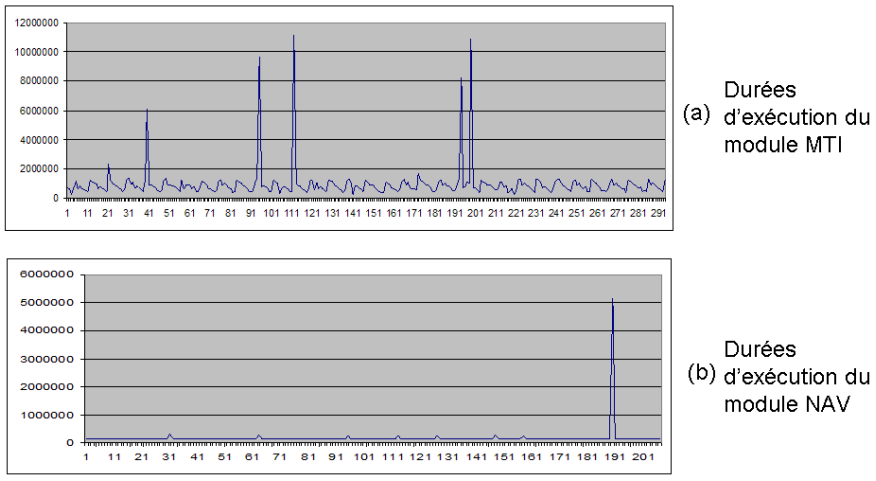


Figure 11.10 – Evolution de la durée d'exécution du module NAV et du module GSV

Résumé du chapitre 11 :

- Nous avons mené des expérimentations dans un lac avec l'AUV du LIRMM Taipan 300 qui ont consisté à suivre une trajectoire rectiligne à des profondeurs différentes.
- L'exploitation des logs de la mission permet d'analyser le comportement de la commande utilisée.
- En observant les activités des modules consignées dans les logs de l'ordonnanceur nous pouvons constater que les différents modules ont été activés dans le bon ordre.

Conclusions & Perspectives

Notre travail traite de l'architecture logicielle de contrôle pour les AUV (Autonomous Underwater Vehicle). La proposition que nous avons formulée est une architecture logicielle de contrôle structurée selon un modèle simple et clair, fondée sur des entités de base appelées modules. Ces modules sont manipulables indépendamment (comportement interne), composables dynamiquement et facilement (de façon intuitive, vus comme des "boîtes noires") par l'intermédiaire de leurs ports de données (data et événements) selon un modèle d'interaction de type producteur/consommateurs et un mécanisme de souscription dynamique. Ces modules sont également paramétrables et contrôlables, respectivement à l'aide de ports de paramétrage et de contrôle. Cette construction de l'architecture de contrôle du robot, fondée sur un assemblage simple des entités de base que sont les modules, répond aux critères visés de modularité, d'évolutivité et de réutilisabilité notamment dans notre contexte d'une flottille d'AUV (réutiliser notre architecture et ses entités sur plusieurs véhicules). Cette composition dynamique est le support de la gestion contextuelle de tâches que nous souhaitions mettre en œuvre au regard du besoin d'autonomie décisionnelle des AUV. En effet, l'exécution effective de cette prise de décision, résultant de la conciliation de l'intention (mission) et de l'obligation (contraintes), tel un changement d'objectif par exemple, impose de pouvoir dynamiquement restructurer (au sens reconfigurer la composition) les modules bas niveau.

L'architecture proposée, s'inscrivant dans les architectures dites mixtes, est structurée selon deux niveaux, à l'image des tendances architecturales, qui correspondent à deux "mondes" différents : le monde événementiel et le monde périodique, avec des fonctionnements et des espaces de représentation différents. Ces niveaux sont d'une part un niveau décisionnel et d'autre part un niveau exécutif, l'articulation entre ces niveaux étant assurée par un ordonnanceur. Le niveau décisionnel supporte une supervision multiple, au sens d'une supervision raffinée : un superviseur global assurant la gestion de la mission et chargé de la gestion des modes et des ressources, et des superviseurs locaux dédiés chacun à la gestion d'une ressource selon un mode de fonctionnement donné (autonome, téléopération, coopération). D'autres raffinements du niveau décisionnel sont possibles, au même titre que ce niveau pourrait être complété par des "fonctionnalités" plus avancées tel qu'un planificateur ou un système délibératif plus évolué[51]. Le niveau exécutif est quant à lui constitué d'un ordonnanceur pilotant l'ensemble des modules bas-niveau, i.e. les modules implémentant les différents traitements relatifs à la gestion de l'instrumentation, au calcul des lois de commande ou de la navigation, etc. L'ordonnanceur qui

contrôle entièrement l'activité des modules, permet d'appliquer les décisions du haut niveau exprimées en terme d'objectifs, en répondant aux différentes contraintes dont celles temps-réel (respect des périodes d'échantillonnage, des dates d'exécution des tâches, etc.) et celles relatives à la gestion des ressources embarquées. Cette approche permet en effet de réifier la gestion des contraintes d'exploitation de l'instrumentation embarquée, dont notamment les contraintes issues de l'interférence potentielle entre capteurs acoustiques.

Pour permettre une prise en main de l'architecture par des acteurs aux compétences et intérêts scientifiques différents, et pour favoriser tant son exploitation que son évolution, la "configuration" de l'architecture est basée sur un vocabulaire simple à manipuler. Ce vocabulaire comprend trois types de termes : les objectifs, les sous-objectifs et les modules. Ces termes expriment respectivement l'intention (objectifs), les capacités du système (modules) et la manière dont les intentions vont être réalisées grâce aux capacités du robot (les sous-objectifs). Ces termes sont décrits à travers des bases de données dédiées et facilement manipulables par les acteurs afin de "configurer" (au sens de décrire et de faire évoluer) l'architecture.

L'exécution d'une mission s'appuie directement sur ces termes (i.e. sur ces bases de données). Au plus haut niveau, le superviseur global est en charge de la gestion de la mission à un niveau stratégique. Les décisions relatives à la planification des tâches robotiques à lancer (lancement à des dates ou à des conditions précises) sont prises à ce niveau. Il décompose la mission reçue de l'opérateur en une séquence d'objectifs envoyés au superviseur local. Le rôle d'un superviseur local est de vérifier la disponibilité des ressources, de réagir aux événements (e.g. évitement d'obstacle), et de subdiviser chaque objectif en sous-objectifs avant de les envoyer en exécution via l'ordonnanceur. Le niveau exécutif, avec comme point d'entrée l'ordonnanceur, fonctionne de manière périodique. Plusieurs modules dans ce niveau sont coordonnés pour configurer les capteurs, calculer les lois de commandes et gérer les conflits liés à l'instrumentation. Pour assurer le respect des contraintes bas niveau (e.g. capteurs acoustiques interférents) l'ordonnanceur gère en ligne toute l'activité des modules bas niveau. Il est une entité à part entière de l'architecture ; entité dont nous avons proposé un algorithme pour gérer les contraintes mentionnées. A l'image de l'évolutivité de l'architecture, cet algorithme peut être modifié sans remettre en cause l'architecture.

L'accessibilité et l'acceptation de cette architecture sont des critères qui ont été considérés dans notre proposition qui devait nécessairement être concrétisée, i.e. disponible à l'issue de la thèse pour les acteurs de l'équipe. Le processus de développement se devait d'être de complexité limitée et la proposition accompagnée d'un ensemble d'outils logiciels permettant sa mise en oeuvre. Nous avons ainsi développé une pseudo-infrastructure, en quelque sorte une couche d'abstraction du système d'exploitation temps-réel, appelée "libmodule", assurant toute la gestion des aspects processus et communication inter-processus selon le modèle défini (producteur/consommateurs avec principe de souscription dynamique). Nous avons également mis en place les services nécessaires à la manipulation des entités (modules) constituant l'architecture et à son "déploiement" sur la carte processeur embarquée dans le véhicule. Enfin, nous avons développé l'outil informatique permettant de visualiser toutes les traces (logs divers tels que les variables échangées, l'activité des modules, etc.) que peut mémoriser le contrôleur lors de l'exécution de la mission.

Ce travail a été conduit jusqu'à son implantation et son expérimentation. Les expérimentations ont été effectuées en milieu lacustre avec le robot Taipan 300 du LIRMM.

Elles nous ont permis de valider le bon fonctionnement de l'architecture logicielle, même si bien évidemment seule une utilisation intensive, à travers de multiples expérimentations, nous permettra de réellement identifier les limitations de notre proposition.

Comme nous venons de le mentionner en conclusion, la première perspective à ce travail de thèse est de réaliser une multitude d'expérimentations pour identifier les limitations de nos travaux. Mais les travaux présentés doivent être perçus comme une base qui peut, et doit, être approfondie en différents points :

- Améliorer l'ordonnanceur, véritable articulation au sein de l'architecture :
  - Au sein de l'algorithme d'ordonnancement, lors du calcul de la priorité des tâches (modules), nous devons prendre en compte les degrés de compatibilité de la tâche considérée (avec les autres tâches éligibles). Ce degré de compatibilité est donné par la taille de la clique qui implique cette tâche dans le graphe de compatibilité. Cela permettra de favoriser la parallélisation des tâches d'acquisition.
  - Pour augmenter la robustesse de l'architecture, nous envisageons de recourir à une pseudo-redondance logicielle. C'est à dire pouvoir exprimer des alternatives dans la décomposition des sous-objectifs en séquences de modules. Ces alternatives concernent les modules à utiliser en cas de "défaillance" des modules nominaux, i.e. en cas de retard conséquent voire même de blocage d'un ou plusieurs modules impliqués dans la séquence nominale (correspondant à un sous-objectif donné). La décision d'exécution d'une alternative, i.e. le choix d'un module de remplacement et répondant bien sûr aux contraintes (par exemple produisant au moins les mêmes variables que celles attendues), relève de la compétence de l'ordonnanceur et reste transparente pour les niveaux décisionnels... sous réserve bien sûr que le module alternatif satisfasse les contraintes temporelles. A défaut, le niveau décisionnel sera interpellé via une notification d'échec.
  - Les interactions entre l'ordonnanceur et le niveau décisionnel doivent être affinées et clarifiées. Il est nécessaire de classifier les différents types d'échec possibles ainsi que d'identifier clairement le superviseur à interpeller (global ou local, i.e. remise en cause de l'objectif ou seulement du sous-objectif) selon la situation.
- Projeter la formulation des entités de base, que sont les modules, vers une formulation plus avancée comme, par exemple, étudier l'apport d'un formalisme basé sur les composants logiciels et leur interconnexion par des connecteurs [19]. Pour des considérations d'analysabilité, il serait d'ailleurs intéressant de fonder (spécifier et exécuter) le comportement interne des modules sur un modèle formel tel que par exemple les réseaux de Petri, qu'il s'agisse des modules bas-niveau ou de ceux de la supervision [52][53]. Enfin, dans la même idée d'une plus grande "standardisation", il faudrait examiner les diverses propositions sur le sujet (mentionnées dans le manuscrit) pour au moins "mettre dans un format standard" les entités échangées au sein de l'architecture (données, messages, événements, ...).
- Développer des outils plus évolués (et basés sur des interfaces plus conviviales) facilitant l'utilisation de l'architecture sur tous les plans : modifications à différents niveaux, mise à jour, déploiement, exploitation des traces (logs), etc..
- Sur un plan plus applicatif, il serait sans aucun doute intéressant d'étudier plus en profondeur la téléopération et la coordination de flottille :
  - Seul le mode de téléopération en surface a été abordé à travers la thèse, mais non exposé dans le manuscrit (et non expérimenté). Il permet un rapatriement

du véhicule, à vue ou non, vers le point de récupération (la rive ou le bateau). Notre architecture permet d'aisément rajouter des modes de fonctionnement ; la téléopération sous-marine pourrait être étudiée et ce à différents niveaux : téléopération haut-niveau basée sur l'envoi d'objectifs à accomplir, ou téléopération bas-niveau basée sur l'envoi de consignes (de position) à suivre. Dans ce dernier cas, il serait intéressant de s'appuyer sur des approches appliquées en terrestre par exemple, comme celle basée sur une adaptation de la commande en fonction de l'évolution de la qualité du lien (acoustique dans notre cas)[54].

– Le cas de la flottille hétérogène (plusieurs véhicules sous-marins et/ou de surface). Dans le cadre de la navigation en flottille, parmi les problèmes posés figurent ceux relatifs aux interférences entre les instrumentations des différents véhicules. Ces interférences affectent aussi la communication inter-véhicules qui est primordiale dès lors que l'on adresse la problématique de la coordination. Notre approche donne une existence explicite à la séquence de modules et permet de générer à l'avance l'échéancier traduisant entre autres les dates précises d'utilisation de l'instrumentation acoustique. Ces informations pourraient être exploitées dans le cadre de la coordination des activités des véhicules (dont la communication), sous réserve de les abstraire suffisamment pour faire face à la limitation de débit des communications acoustiques. Il faut étudier la nature de l'information requise et les connaissances "croisées" nécessaires, au regard du but de la coordination. A titre d'exemple, transmettre seulement la liste des sous-objectifs en cours (sous forme "compactée") peut s'avérer suffisant si l'on suppose que le véhicule récepteur a la capacité d'en déduire (par simulation) l'ordonnancement (i.e. l'échéancier).

Bibliographie

[1] K. Capek. *R.U.R. (traduction française 1997).* Editions de l'Aube, 1920.

[2] Bernard Espiau. La peur des robots. *Science revue*, Hors série n°10 :page 49, mars 2003.

[3] F. Kaplan. Un robot peut-il être notre ami ? In Y. Orlarey, editor, *L'Art, la pensée, les émotions*, pages 99–106. Grame, 2001.

[4] L. Fournet. Positionnement d'engins autonomes grands fonds : Etat de l'art et perspectives. *Mémoire ingénieur E.S.G.T*, pages 6–10, 2002.

[5] JAUS Working Group. Joint Architecture for Unmanned Systems JAUS, Reference Architecture Specification, volume 2, v.3.2. 2004.

[6] Aurélien Godin. Pleading for open modular architectures. *Control Architecture of Robots - CAR'06*, 2006.

[7] E. Danson. The economics of Scale : Using Autonomous Underwater Vehicles AUVs for Wide-Area Hydrographic Survey and Ocean Data Acquisition. *Proceedings of XXII FIG(Fédération Internationale des Géomètres) International Congress, Washington, D.C. USA*, April 19-26 2002.

[8] Jean-Claude Michel. La mine de demain : une exploitation sous-marine durable. *Géochronique*, mars 2006.

[9] L. Lapierre. Underwater Robots Part I : Current Systems and Problem Pose. *Mobile Robotics - Towards New Applications*, 2007.

[10] Fernando J. A. S. Barriga Yves Fouquet Myriam Sibuet Pierre-Marie Sarradin Patrick Siméoni Jean-François Drogou Jean-Louis Michel, Michaël Klages. Victor 6000 : Design, Utilization and First Improvements. *Proceedings of The Thirteenth (2003) International Offshore and Polar Engineering Conference*, May 2003.

[11] Bjorn Jalving, Jon Kristensen, and Nils Storkersen. Program Philosophy and Software Architecture for the HUGIN Seabed Surveying UUV. *Proc. Control Applications in Maritime Systems (CAMS-98), Fukuoka, Japan*, October 1998.

[12] Roar Marthiniussen, Karstein Vestgard, and Rolf Ame Klepaker. HUGIN-AUV Concept and operational experiences to date. *OCEANS'04, Kobe, Japan*, 2004.

[13] A. Pascoal, P. Oliveira, C. Silvestre, L. Sebastiao, M. Rufino, V. Barroso, J. Gomes, G. Ayela, P. Coince, M. Cardew, A. Ryan, H. Braithwaite, N. Cardew, J. Trepte,

N. Seube, J. Champeau, P. Dhaussy, V. Sauce, R. Moitie, R. Santos, F. Cardigos, M. Brussieux, and P. Dando. Robotic ocean vehicles for marine science applications : the european ASIMOV project. In *OCEANS 2000 MTS/IEEE Conference and Exhibition*, volume 1, pages 409–415, Providence, RI, USA, 2000.

[14] A.J. Healey D.B. Marco and R.B. McGhee. Autonomous Underwater Vehicles : Hybrid Control of Mission and Motion. In *Autonomous Robots 3, 169-186*, 1996.

[15] *International Workshop on Software Development and Integration in Robotics : Understanding Robot Software Architectures (ICRA), Barcelone, Espagne*, April, 18-22 2005.

[16] *Second International Workshop on Software Development and Integration in Robotics : Understanding Robot Software Architectures (ICRA), Rome, Italy*, April, 14.

[17] *Control Architecture of Robots (CAR'06), Montpellier, France*, April, 6-7.

[18] *Control Architecture of Robots (CAR'07), Paris, France*, May 31 and June 1st.

[19] R. Passama. *Conception et développement de contrôleurs de robots - Une méthodologie basée sur les composants logiciels*. PhD thesis, Université Montpellier II, Laboratoire d'Informatique de Robotique et de Microélectronique de Montpellier (LIRMM), 2006.

[20] Rodney A. Brooks. A robust layered control system for a mobile robot. *Artificial intelligence at MIT : expanding frontiers, Cambridge, MA, USA*, pages 2–27, 1990.

[21] J. K. Rosenblatt. DAMN : A distributed architecture for mobile navigation. In *Proc. of the AAAI Spring Symp. on Lessons Learned from Implememted Software Architectures for Physical Agents*, Stanford, CA, 1995.

[22] E. Gat. On three-layer architecture. *A.I. and mobile robots, D. Korten Kamp & al. Eds. AAAI Press*, 1998.

[23] J.S. Albus, R. Lumia, J. Fiala, and A. Wavening. Nasrem : The NASA/NBS standard reference model for telerobot control system architecture. *Technical report, Robot System Division, National Institute of Standards and Technologies*, 1997.

[24] J.S. Albus. 4D/RDC : A reference model architecture for unmanned vehicle systems. *Technical report, NISTIR 6910*, 2002.

[25] A. Tate, B. Brabble, and R. Kirby. O-plan2 : an open architecture for command, planning and control. *Technical report, Artificial Intelligence Application Institute, University of Edimburg*, 1994.

[26] S. Lemai, B. Diaz, and N.Muscettola. A real-time rover executive based on model-based reactive planning. *Proc. of International Conference on Advanced Robotics (ICAR'03), Coimbra, Portugal*, 2003.

[27] R. Alami, R. Chatila, S. Fleury, M. Ghallab, and F. Ingrand. An architecture for autonomy. *International Journal of Robotics Research*, 1998.

[28] S. Fleury, M. Herrb, and R. Chatila. Genom : A tool for the specification and the implementation of operating modules in a distributed robot architecture. *International Conference on Intelligent Robots and Systems (IROS'97)*, Septembre 1997.

[29] R. Volpe, I. Nesnas, T. Estlin, D. Mutz, R. Petras, and H. Das. Claraty : Coupled layer architecture for robotic autonomy. *Technical report, Jet Propulsion Laboratory*, 2000.

[30] R. Volpe, I. Nesnas, T. Estlin, D. Mutz, R. Petras, and H. Das. The claraty architecture for robotic autonomy. *IEEE Aerospace Conference (IAC-2001)*, Mars 2001.

[31] R. Volpe, I. Nesnas, T. Estlin, D. Mutz, R. Petras, and H. Das. Toward developping reusable software components for robotic applications. *International Conference on Intelligent Robots and Systems*, October 2001.

[32] C. Urmson, R. Simmons, and I. Nesnas. A generic framework for robotic navigation. *IEEE Aerospace Conference (IAC-2003)*, Mars 2003.

[33] P. Oliveira, A. Pascoal, V. Silva, and C. Silvestre. Design, Development, and Testing at Sea of the Mission Control System for the MARIUS Autonomous Underwater Vehicle. *International Journal of Systems Science*, 1997.

[34] A. Pascoal. The AUV MARIUS : Mission scenarios, Vehicle Design, Construction and Testing. *Proc. Of 2nd workshop on mobile robots for subsea environments, Monterey Bay Aquarium, Minterey, California, USA*, May 1994.

[35] F. Cottet, J. Delacroix, C. Kaiser, and Z. Mammeri. Sheduling in Real-Time Systems. *John Wiley and Sons*, 2002.

[36] Xavier Blanc, Jérôme Delatour, and Tewfik Ziadi. Benefits of the MDE approach for the development of embedded and robotic systems. *Control Architecture of Robots CAR'07, Paris, France*, May 31 and June 1st.

[37] C. Riquier, N. Ricard, and C. Rousset. DES Data Exchange System, a publish/subscribe architecture for robotics. *Control Architecture of Robots CAR'06, Montpellier, France*, April, 6-7 2006.

[38] A. El Jalaoui, D. Andreu, and B. Jouvencel. Architecture de control pour la gestion contextuelle de tâches sur un AUV. *Journées Nationales de Rechercher en Robotique (JNRR'05), Guidel, France*, october 2005.

[39] A. El Jalaoui, D. Andreu, and B. Jouvencel. A control software architecture for contextual tasks management : application to the AUV Taipan. *OCEANS'05, Brest, France*, june 2005.

[40] A. El Jalaoui, D. Andreu, and B. Jouvencel. AUV Control Architecture for control Management of Embedded Instrumentation. *4th IFAC Symposium on Mechatronic Systems MECHATRONICS, Heidelberg, Germany*, september 2006.

[41] M. Garey and D. Johnson. Complexity results for multiprocessor scheduling under ressouce constraints. *Journal of Computing*, 1975.

[42] A. El Jalaoui, D. Andreu, and B. Jouvencel. Contextual Management of Tasks and Instrumentation within an AUV control software architecture. *International Conference on Intelligent Robots and Systems (IROS'06), Beijing, China*, october 2006.

[43] Simonin Gilles. *Complexité et solutions algorithmiques pour des problèmes d'ordonnancement appliqués a l'acquisition de données pour une torpille en immersion.* Mémoire Master, Université Montpellier II, Laboratoire d'Informatique de Robotique et de Microélectronique de Montpellier (LIRMM), 2006.

[44] J.M. Spiewak, B. Jouvencel, and P. Fraisse. A New Design of AUV for Shallow Water Applications : H160. *International Offshore and Polar Engineering (ISOPE'06)*, 2006.

[45] L. Lapierre, V. Creuze, and B. Jouvencel. Robust diving control of an AUV. *Manoeuver and Control of Marine Craft (MCMC'06), Lisbon, Portugal*, 2006.

[46] J. Vaganay, B. Jouvencel, P. Lepinay, and R. Zapata. Taipan an AUV for very shallow water applications. *World Automation Congres*, 1998.

[47] J.M. Spiewak. *Contribution à la Coordination de flottille de Véhicules sous-marins autonomes.* PhD thesis, Université Montpellier II, Laboratoire d'Informatique de Robotique et de Microélectronique de Montpellier (LIRMM), 2007.

[48] Manoël Trapier. *Mise en oeuvre 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.

[49] O. Parodi, A. El Jalaoui, and B. Jouvencel. Thetis : A real-time Multi-Vehicles Hybrid Simulator for Heterogeneous Vehicles. *submitted to the 17th IFAC World Congress*, 2008.

[50] O.Parodi, A. El Jalaoui, and D. Andreu. Connectivity of Thetis, a Distributed Hybrid Simulator, with a Mixed Control Architecture. *4th International Conference on Autonomic and Autonomous Systems*, 2008.

[51] S. Lacroix, A. Mallet, D. Bonnafous, G. Bauzil, S. Fleury, M. Herrb, and R. Chatila. Autonomous Rover Navigation on Unknown Terrains : Functions and Integration. *International Journal of Robotics Research, 21(10-11), pages 917-942*, 2002.

[52] M. Barbier, J.F. Gabard, D. Vizcaino, and O. Bonnet-Torres. ProCoSA : a software package for autonomous system supervision. *First Workshop on Control Architectures of Robots (CAR'06), Montpellier, France*, April 2006.

[53] Caccia M., Coletta P., Bruzzone G., and Veruggio G. Petri net-based execution control of robotic tasks. *Mediterranean Conference on Control and Automation, Dubrovnik, Kroatia*, June 2001.

[54] P. Fraisse. Commande de Robots à Architectures Complexes. *Habilitation à Diriger les Recherches, Université Montpellier II*, 2004.

[55] D. Andreu. *Commande et Supervision des Procédés Discontinus : une Approche Hybride.* PhD thesis, Université Paul Sabatier, Laboratoire d'Analyse et d'Architecture des Systèmes (LAAS), 1996.

[56] E. Dombre and W. Khalil. *Modélisation et commande des robots.* Hermes, 1988.

[57] F. Ingrand. Architecture Logicielles pour la Robotique Autonome. In *Journées Nationales de Rechercher en Robotique (JNRR'03)*, octobre 2003.

[58] A.A.D de Medeiros, R. Chatila, and S. Fleury. Specification and validation of a control architecture for autonomous mobile robots. *IEEE International Conference on Itelligent Robots and Systems (IROS'96)*, Novembre 1996.

[59] A. J. Healey, D. B. Marco, P. Oliveira, A. Pascoal, V. Silva, and C. Silvestre. Strategic Level Mission Control - An Evaluation of CORAL and PROLOG Implementations for Mission Control Specifications. In *Proceedings of the 1996 Symposium on Autonomous Underwater Vehicle Technology, pp.125-132, Monterey California*, 1996.

[60] V. Silva C. Silvestre P. Oliveira, A. Pascoal. On the Design and Development of Mission Control Systems for Autonomous Underwater Vehicles : An Application to the Marius AUV. In *The Sixth International Symposium on Robotics and Automation*, 1996.

[61] V. Silva C. Silvestre P. Oliveira, A. Pascoal. Design, Development, and Testing at Sea of the Mission Control System for the MARIUS Autonomous Underwater Vehicle. In *The Sixth International Advanced Robotics Program IARP-96*, 1996.

[62] V. Silva C. Silvestre P. Oliveira, A. Pascoal. Mission Control of the MARIUS AUV : System Design, Implementation, and Sea trials. In *International Journal of System Sciences, pp.1065-1080*, 1998.

[63] Jérôme Lemaire Claude Barrouil. Advanced Real-time Mission Management for an AUV. In *Advanced Mission Management end Sytem Integration Technologies for Improved Tactical Operations, Florence, Italy*, 1999.

[64] Jérôme Lemaire Claude Barrouil. An integrated navigation system for a long range AUV. In *OCEANS'98*, 1998.

[65] R.B. McGhee A.J. Healey, D.B. Marco. Autonomous Underwater Vehicle Control Coordination Using A Tri-Level Hybrid Software Architecture. In *IEEE, International Conference on Robotics and Automation*, 1996.

[66] Mike Campbell Duane Davis Tony Healey Mike Holden Brad Leonhardt Dave McClarin Bob McGhee Don Brutzman, Mike Burns and Russ Whalen. NPS Phoenix AUV Software Integration and In-Water Testing. In *Symposium on Autonomous Underwater Vehicle Technology,pp.99-108, Monterey California*, 1996.

[67] J.Batlle K.Sugihara P.Ridao, J.Yuh. On AUV Control Architecture. In *Proceedings of the International Conference on Robots and Systems*, 2000.

Thèse de Abdellah EL JALAOUI

Gestion Contextuelle de Tâches pour le contrôle d'un véhicule sous-marin autonome

Les travaux présentés s'inscrivent dans le cadre des architectures logicielles de contrôle des AUV (Autonomous Underwater Vehicle). La commande d'un AUV est basée sur un ensemble de ressources informatiques embarquées et un ensemble de capteurs/actionneurs qui peut changer selon le type de mission confiée au robot (modification de la charge utile). Le robot est comparable à un "porte charge" adaptable à différentes tâches et par conséquent évoluant au grés des progrès technologiques et de l'apparition de nouvelles applications scientifiques.

De plus, le besoin d'autonomie dans un environnement (milieu sous-marin) en constante évolution et souvent inconnu requiert de la part du véhicule d'être capable, à chaque moment, d'évaluer son état et l'état de son environnement afin de prendre les décisions cohérentes pour exécuter sa mission. La réalisation d'un tel véhicule autonome requiert une méthodologie de conception de son architecture logicielle/matérielle.

Ici, nous présentons l'architecture développée au LIRMM pour l'AUV Taipan. Cette architecture est construite en respectant certains critères tels que la modularité, l'évolutivité et la réutilisabilité.

L'architecture est construite à partir de composants indépendants appelés "modules". Ces modules possèdent un ensemble de ports entrée/sortie qui vont permettre l'établissement dynamique de flux de données/contrôle. Il est alors possible de programmer les modules séparément, de les modifier et de les tester avant de les assembler.

L'architecture mixte proposée repose sur deux niveaux hiérarchiques : un niveau décisionnel comprenant un Superviseur Global et des Superviseurs Locaux (un pour chaque mode : autonome, téléopération, coopération) et un niveau exécutif basé sur un ordonnanceur et des modules.

Un vocabulaire basé sur trois types de termes : les objectifs, les sous-objectifs et les modules est utilisé au sein de l'architecture pour exprimer l'intention (objectifs), les capacités du système (modules) et la manière dont les intentions vont être réalisées grâce aux capacités du robot (sous-objectifs).

Le plus haut niveau, le superviseur global, est en charge de la gestion de la mission à un niveau stratégique. Les décisions relatives aux tâches robotiques à lancer à des dates précises, la planification de ces tâches sont prises à ce niveau. Il décompose la mission reçue de l'opérateur en une séquence d'objectifs envoyés au superviseur local. Le superviseur local vérifie la disponibilité des ressources, réagit aux événements immédiats (e.g. évitement d'obstacle). Il subdivise chaque objectif en sous-objectifs avant de les envoyer au niveau exécutif. Le niveau exécutif fonctionne de manière périodique. Plusieurs modules dans ce niveau sont coordonnés pour configurer les capteurs, calculer les lois de commandes et gérer les conflits liés à l'instrumentation. Pour assurer le respect des contraintes bas niveau (e.g. capteurs acoustiques interférents) un ordonnanceur est en charge de gérer en ligne l'activité des modules bas niveau.

Pour illustrer les principaux aspects de notre approche, un exemple d'application a été développé et testé sur l'AUV Taipan.

Mots clés : robotique sous-marine, AUV, architecture logicielle de contrôle, approche contextuelle, gestion de l'instrumentation.

Contextual Management of Tasks for the control of an autonomous underwater vehicle

The presented work belongs to the field of AUV (Autonomous Underwater Vehicle) control software architecture. The control of an AUV is based on a set of embedded computer resources and a set of sensors/actuators which can change according to the robot mission type (change of payload for instance). The robot should be comparable to a general-purpose vehicle adaptable to different tasks, and consequently evolving in accordance with the technological progress or the appearance of new challenging scientific applications.

Moreover, the need for autonomy in an environment in constant evolution and frequently unknown requires on behalf of the vehicle to be able, at every moment, to evaluate its state and the state of its environment, in order to make coherent decisions while executing the mission. The realization of such type of autonomous vehicle requires a methodology of design of its embedded software/hardware architecture.

Here, we present the software architecture developed at LIRMM for the AUV TAIPAN. This architecture is built respecting several criterions like modularity, evolutionarity and reusability.

The architecture is made up of independent components called "modules". Modules have a set of ports that permits data/control flows to be dynamically established between them. It is then possible to program the modules separately, to modify them and to test them before assembling them.

This proposed mixed architecture is organised on a hierarchical basis according to two levels : a decisional level containing a Global Supervisor and Locals Supervisors (one for each mode : autonomous, teleoperation, cooperation) and an Executive Level based on a scheduler and a set of modules. A vocabulary based on three types of terms : objectives, sub-objectives, and modules is used within the architecture to express the intention (objectives), the system capacity (modules) and the way to achieve the intention thanks the robot capacity (sub-objectives).

The higher level, the global supervisor, is in charge of strategic mission management. Decisions related to the robotic tasks (objectives) to launch at a precise time and to mission planning are taken within this level. It decomposes the mission received from the operator into a sequence of objectives sent to the local supervisor. A local supervisor checks resources availability, reacts to events which require fast response (e.g. obstacle avoiding). It divides each objective in sub-objectives before sending them to the executive level. This executive level works in a periodical way. Several modules within this level are coordinated to configure sensors, to compute controls, to manage instrumentation conflicts. To ensure the respect of low level constraints (e.g. acoustical interfering sensors) a scheduler is in charge of low level modules activity management.

To illustrate main aspects of our approach, an application example is developed and tested on the AUV Taipan.

Keywords : underwater robotics, AUV, control software architecture, contextual approach, instrumentation management.