Exploration robotique de l'environnement aquatique : les modèles au coeur du contrôle

Adrien Lasbouygues

THÈSE

Pour obtenir le grade de

Docteur

Délivré par l'**Université de Montpellier**

Préparée au sein de l'école doctorale **I2S**
Et de l'unité de recherche **LIRMM**

Spécialité: **Génie Informatique et Automatique**

Présentée par **Adrien LASBOUYGUES**

Exploration robotique de l'environnement aquatique : les modèles au cœur du contrôle

Soutenue le 10 Décembre 2015 devant le jury composé de

Mr. Jacques Malenfant	Professeur	LIP6	Rapporteur
Mr. Benoit Zerr	Maître de Conférence, HDR	ENSTA	Rapporteur
Mr. Simon Lacroix	Directeur de Recherche	LAAS	Examinateur
Mr. René Zapata	Professeur	LIRMM	Examinateur
Mr. David Andreu	Maître de Conférence, HDR	LIRMM	Directeur de thèse
Mr. Lionel Lapierre	Maître de Conférence	LIRMM	Co-Encadrant de thèse
Mr. Hervé Jourde	Professeur	HSM	Invité

"Life is about passions. Thank you for sharing mine"

Michael Schumacher

Remerciements

Une thèse est un long et grand voyage de trois ans et surtout une grande aventure humaine faite de nombreuses rencontres et de multiples échanges.

Tout d'abord, je tiens à remercier tous les membres du LIRMM pour leur accueil et leur disponibilité, avec une mention spéciale pour Nicolas Serrurier pour m'avoir guidé dans le labyrinthe de la bureaucratie et des nombreux documents administratifs à remplir.

Je souhaite aussi remercier tous les membres de l'équipe Explore, Karen, René, Jean, Bruno, Didier, Sébastien et Pascal pour leur accueil et leur aide tout au long de ma thèse.

Je veux également remercier Robin Passama, le grand maître Yoda de ContrACT, pour ses suggestions qui ont toujours été pertinentes et m'ont aidé à orienter mes travaux dans la bonne direction et pour ses conseils qui toujours pleins de sagesse ont été. Un grand merci aussi pour sa disponibilité et son aide dans la compréhension d'un ContrACT parfois un tantinet capricieux.

Je remercie mon directeur de thèse David Andreu, pour son aide, son soutien et sa franchise tout au long de mon parcours. Ainsi, David m'a aidé à toujours me remettre en question afin d'avancer dans la bonne direction. Merci aussi à lui d'avoir patiemment relu beaucoup, beaucoup de pages et je pense que je te dois un bon paquet de cafés. Je souhaite également remercier Lionel Lapierre mon co-encadrant de thèse pour ses nombreux conseils, pour avoir partagé avec moi un peu de sa grande expérience de l'automatique et puis pour avoir, lui aussi, lu vraiment beaucoup de pages. Je dois également remercier leurs familles qui ont gentiment accepté, même en plein été, de partager leur piscine avec notre robot afin que nous puissions effectuer les tests nécessaires à l'avancée de nos travaux.

Je souhaite également remercier Messieurs Jacques Malenfant et Benoit Zerr pour avoir accepté la longue et difficile mission qui leur a été confiée, examiner mes travaux de recherche, ainsi que Messieurs René Zapata, Simon Lacroix et Hervé Jourde qui ont accepté de faire partie de mon jury de thèse.

Bien entendu, je veux également remercier tous mes camarades thésards avec qui j'ai partagé ces trois années. Ceux qui ont déjà fini, Divine, Nicolas, Johann, David, Florent et Arnaud. Ceux qui comme moi vont en terminer avec cette aventure, Ederson, Moussab, Ahmed,

Mohammed. Ceux pour qui le travail continue, Irina, Yadpiroon, Gamal et Lotfi. Je souhaite également remercier ceux qui ont travaillé avec moi sur ce projet et notamment Benoit Ropars avec qui j'ai partagé bons moments et galères trois années durant mais également Maxime Rendu qui fut un fantastique stagiaire et Silvain Louis d'abord en temps que stagiaire puis comme thésard. Silvain, je te souhaites beaucoup de courage pour poursuivre le projet sur le Jack.

Un merci tout particulier à Nicolas, un ami comme il en existe peu, pour nos discussions sur tout et n'importe quoi (souvent n'importe quoi d'ailleurs), le genre d'échanges qui aident toujours à garder la tête froide et beaucoup de bonne humeur même le plus maussade des Lundi.

Il faut également que je remercie ma fidèle Ford Focus qui nous a, trois ans durant, transportés sans rechigner aux quatre coins du département afin que nous puissions tester et expérimenter dans tous les points d'eau à notre portée. Et en plus, elle a un coffre parfaitement adapté au Jack, ce qui s'est avéré fort pratique.

Je souhaiterais également remercier toute ma famille pour son soutien constant durant ces trois ans en m'ayant aidé, même dans les moments les plus compliqués, à me recentrer sur les choses essentielles de la vie. Et évidemment merci mille fois à ma mère et mon père sans qui je n'en serais certainement pas là aujourd'hui et qui m'ont toujours soutenu durant cette thèse même si je n'étais pas tous les jours facile à vivre.

Il ne me reste plus qu'à souhaiter au lecteur un bon voyage et surtout une bonne lecture.

Table des matières

Table des figures

Liste des tableaux

Introduction

Le terme *robot* fut introduit en 1920 par l'écrivain tchèque Karel Capek dans sa pièce de théâtre R.U.R. (Rossum's Universal Robots). Si les robots actuels sont bien différents de ceux imaginés par Capek, ils conservent leur principale caractéristique à savoir le fait de remplacer des humains dans des tâches pénibles ou dangereuses. Ainsi, les premiers robots furent conçus, à partir des années 40, tout d'abord pour permettre la manipulation à distance de matériel nucléaire puis pour la production sur des chaînes automatisées. Ils évoluent dans un environnement connu et ne disposent d'aucune capacité de décision, répétant des actions programmées par un opérateur humain.

Néanmoins d'autres robots furent dotés de la capacité de se déplacer dans leur environnement tout en étant équipés de capteurs leur permettant d'interagir avec celui-ci. Développé à la fin des années 40 par le neurophysiologiste William Grey Walter, le robot tortue *Machina speculatrix*, surnommé Elsie, fut l'un des tous premiers robots mobiles [Wal50]. C'est à partir des années 60 que la robotique mobile commença réellement à se développer avec, par exemple, le robot "Beast" [WS05] développé à l'université John Hopkins puis le robot Shakey (Université de Standford, à partir de 1966) [Nil69] qui fut l'un des premiers robots à intégrer et permettre le développement des travaux sur l'Intelligence Artificielle. Néanmoins les limitations sur la perception de l'environnement à l'époque imposaient de faire évoluer ces robots dans un environnement fortement structuré.

Puis, les progrès technologiques réalisés à la fois sur les unités de calcul, dont la performance s'accroit tandis que la taille se réduit, les moyens de stockage de l'énergie et les capteurs ainsi que les avancées des disciplines impliquées dont l'automatique ou l'informatique ont permis aux robots d'évoluer dans des environnements de plus en plus complexes, sortant ainsi des laboratoires pour s'aventurer dans le vaste monde, tout en étant toujours plus autonomes.

De fait, et grâce à leur capacité à opérer dans des environnements dangereux pour l'Homme, les robots ont commencé à être utilisés dans de nombreux domaines. Ainsi ils ont, par exemple, trouvé des applications dans l'exploration spatiale, le domaine militaire (approvisionnement sur le champ de bataille, surveillance, déminage) ou encore l'industrie pétrolière (intervention

en grande profondeur sur les plateformes de forage).

De plus, les progrès réalisés ont ouvert la robotique d'exploration à un nouveau public. Ainsi de nombreux scientifiques considèrent les robots comme un vecteur leur permettant d'acquérir des données sur les environnements qu'ils étudient. C'est plus particulièrement vrai dans le cadre de notre étude, la robotique sous-marine. En effet, que ce soit pour l'exploration des eaux terrestres (lacs, rivières, aquifères), des mers ou des océans, la surface à couvrir est vaste et certains environnements (aquifères karstiques, failles océaniques) sont également dangereux voire inaccessibles pour l'Homme. Les scientifiques qui étudient le milieu marin (hydrogéologues, biologistes, archéologues sous-marins, entre autres) considèrent les robots comme un outil de plus en plus pertinent pour réaliser les mesures nécessaires à la compréhension d'environnements souvent hautement complexes.

On peut distinguer deux grandes classes de robots sous-marins. D'un côté, les ROVs (Remotely Operated Vehicles) sont des engins physiquement reliés à la surface, via un "cordon ombilical", alors que les AUVs (Autonomous Underwater Vehicles) sont autonomes. Les ROVs présentent une technologie relativement mature et sont déjà largement utilisés dans l'industrie. Néanmoins, ils sont fortement limités par le lien physique qui les relie à la surface. D'un autre côté les AUVs offrent des perspectives plus étendues en termes d'applications réalisables. En effet, détachés des contraintes de liaison à la surface, ils peuvent effectuer des missions sur des durées et des distances bien plus importantes tout en pouvant se rendre dans des lieux où la présence d'un ombilical est problématique.

Pour ces deux catégories de robots, les recherches visent à accroître leur autonomie opérationnelle. Dans le cadre des ROVs, il s'agit principalement d'assister l'opérateur afin de lui permettre de se concentrer uniquement sur les tâches les plus délicates à réaliser. Pour les AUVs, il s'agit surtout de s'assurer que le robot puisse mener à bien sa mission dans des environnements complexes. Ainsi pour répondre à ces besoins, il est nécessaire de développer de manière de plus en plus poussée les engins robotisés.

Néanmoins concevoir un robot mobile répondant aux besoins d'une application robotique, c'est-à-dire l'application provenant de l'interaction entre un robot, son environnement et la tâche qu'il doit accomplir, est un processus d'une grande complexité nécessitant des expertises variées. Outre la complexité de concevoir l'architecture matérielle du robot (système de locomotion, capteurs, source d'énergie, électronique embarquée), le contrôle du robot est au centre d'une grande attention car son bon fonctionnement joue un rôle clé dans la réussite de sa mission.

Or le logiciel de contrôle correspond à la projection sur une architecture logicielle (temps-réel) d'équations de contrôle établies par les automaticiens. En outre, ces équations intègrent, et souvent construisent, au fil de la mission des connaissances sur l'environnement.

Mais l'utilisation plus large des robots lors de missions scientifiques est, en partie, pénalisée par la difficulté de développer des logiciels de contrôle pour la grande diversité d'applications robotiques possibles et par la complexité d'intégration de la connaissance de l'environnement au sein des lois de commande. L'une des principales raisons à cela est la manière dont sont conçues les lois de commande par les automaticiens. En effet, comme le souligne [UFH+12], ces derniers se concentrent souvent sur la résolution du problème de contrôle laissant de côté toute autre préoccupation. Ainsi les lois de commande proposées sont souvent monolithiques, mélangeant les connaissances et ne laissant pas toujours apparaître les relations entre elles.

Qui plus est, les automaticiens ne prennent que relativement peu en compte le fait qu'un contrôle doit être implémenté sur une architecture logicielle et matérielle qui amène ses propres contraintes (puissance de calcul, échantillonnage des mesures et du pilotage des actionneurs) [UFH+12]. Enfin, la nature monolithique du contrôle limite fortement la marge de manœuvre des ingénieurs logiciels lorsqu'ils doivent implémenter une loi de commande.

Ainsi, concevoir les lois de commande de manière modulaire est une réponse aux problématiques évoquées. En effet, elle devrait pouvoir nous permettre de simplifier la conception des lois de commande entre les différentes applications robotiques en permettant de mutualiser les connaissances communes aux différentes applications tout en clarifiant le rôle que joue chaque connaissance dans le contrôle et en explicitant les relations qui sont établies entre elles. De fait, cela permet de simplifier l'intégration des connaissances spécifiques aux différentes applications. Une plus grande modularité des lois de commande permettrait aussi de laisser plus de liberté aux ingénieurs logiciels dans leur manière de les implémenter.

Il est également nécessaire de définir un terrain commun entre automaticiens et ingénieurs logiciels afin de permettre une meilleure articulation entre les deux domaines autour de préoccupations communes telles que des considérations temporelles. En effet, celles-ci ont un impact majeur sur la stabilité du système contrôlé et fournissent en outre de précieuses informations pour permettre à l'ingénieur logiciel de paramétrer son architecture.

Notre travail a donc pour but de proposer une nouvelle méthodologie permettant d'établir une meilleure synergie entre d'un côté l'automatique et de l'autre le génie logiciel, de la description d'une loi de commande à sa mise en œuvre sur la cible d'exécution. Nous proposons de décrire les lois de commande non plus comme des blocs monolithiques mais comme des compositions de connaissances sur lesquelles nous réifions les contraintes émanant des besoins

de l'automaticien et des capacités de la cible d'implémentation. Nous nous sommes concentrés, dans le cadre de cette thèse, sur les considérations temporelles. Ces contraintes vont nous aider à réaliser la projection de notre loi de commande sur notre cible technologique. Cela permet la vérification de la faisabilité de la projection et la structuration du logiciel de contrôle sur l'architecture logicielle de la cible d'implémentation.

Ce manuscrit est organisé selon quatre parties et dix chapitres, de la manière suivante.

La première partie présente une étude bibliographique. Le premier chapitre évoquera différentes applications de la robotique sous-marine dans le cadre des environnements confinés et faible fond. Le second chapitre présentera les différentes approches pour prendre en compte la modularité en automatique, dans le cadre de la représentation de connaissances et lors de la conception de logiciels de contrôle. Dans le troisième chapitre, nous situerons notre proposition par rapport à ces approches.

Dans la seconde partie, nous expliquerons comment décrire une loi de commande comme une *Composition* d'entités. Tout au long des quatrième et cinquième chapitres, nous donnerons les définitions des différents concepts introduits dans le cadre de cette thèse. Une illustration de ces concepts sur des exemples simples ainsi que l'interface de programmation (API) proposée pour implémenter ces différents concepts seront également présentées en annexe.

La troisième partie traitera de notre approche pour la projection de nos *Compositions* sur une architecture logicielle. Dans le chapitre six, nous développerons le processus consistant à manipuler le graphe associé à une *Composition* pour pouvoir intégrer les contraintes émanant à la fois de l'automatique et de notre implémentation, et ainsi structurer notre contrôle pour faciliter son implémentation. Ensuite, dans le chapitre suivant, nous présenterons le Middleware ContrACT ainsi que la manière d'utiliser notre structuration pour projeter nos entités sur une cible d'exécution.

Dans la quatrième partie, nous présenterons des exemples et expérimentations réalisés pour tester notre approche. Dans le chapitre huit, nous présenterons le robot utilisé et les évolutions qui lui ont été apportées ainsi que les différentes étapes de notre processus d'implémentation qui nous amènent à valider les lois de commande de leur description à leur implémentation. Dans le chapitre suivant, nous illustrerons les apports de la structuration des lois de commande sous forme de *Compositions* ainsi que de méthodologie via l'exemple simple d'un contrôle en cap. Enfin, tout au long du chapitre dix, une fonctionnalité d'évitement de parois en environnement karstique sera développée.

Ce manuscrit s'achèvera par une conclusion ainsi qu'une discussion sur les perspectives ouvertes à l'issue de nos travaux.

Première partie

Contexte et état de l'art

Chapitre 1

Robotique sous-marine et applications

L'eau représente 71% de la surface terrestre. Les milieux aquatiques, qu'ils soient marins ou qu'il s'agisse d'eaux terrestres (lacs, rivières, aquifères), sont un fantastique réservoir de ressources pour l'Homme (minerais, pétrole, alimentation, et l'eau elle-même). Mais si les milieux aquatiques constituent un écosystème d'une formidable richesse, ils sont également très fragiles et d'une grande complexité. Ils sont donc au centre de nombreux enjeux à la fois sociétaux, environnementaux, économiques, militaires et scientifiques. Ainsi de tout temps les hommes ont tenté de repousser leurs limites et d'aller toujours plus loin, levant l'un après l'autre les verrous scientifiques rendant l'impossible possible. L'une des premières problématiques fut de permettre à l'Homme de respirer sous l'eau. Ainsi en 1535, s'inspirant d'un *design* de Léonard de Vinci, Gugliemo de Lorena plonge sur les épaves de deux galions en utilisant une cloche de plongée. D'un principe relativement simple, elle utilise la pression de l'eau afin de maintenir l'air dans une cloche permettant au plongeur de respirer. L'homme comprit rapidement l'intérêt de pouvoir se déplacer sous l'eau pour explorer le milieu marin mais surtout à des fins militaires pour surprendre l'ennemi. Ainsi les cloches, incapables de se déplacer par elles-même, furent rapidement abandonnées au profit d'études pour réaliser un système capable de fournir une réserve d'air tout en se déplaçant sous l'eau. En 1776, David Bushnell conçoit une machine en forme d'œuf constituée de deux coques en bois assemblées ensemble et renforcées par des bandes d'acier. Propulsé par une hélice activée manuellement et capable de plonger en remplissant un petit réservoir d'eau puis en le vidant à l'aide d'une pompe manuelle (ballast), le vaisseau est baptisé *Turtle*. Les sous-marins sont nés. Les développements se poursuivent aboutissant à des engins de taille toujours plus imposante et abandonnant la propulsion manuelle au profit de la propulsion mécanique. Les deux Guerres Mondiales consacrent les sous-marins comme armes de guerre d'une redoutable efficacité. À la fin de la Seconde Guerre Mondiale, les techniques des militaires sont utilisées pour partir à l'exploration des grands fonds. En 1960, Jacques Piccard et Don Walsh atteignent le fond de

la fosse des Mariannes à une profondeur de 10920 mètres à bord du *Trieste* [Lap06].

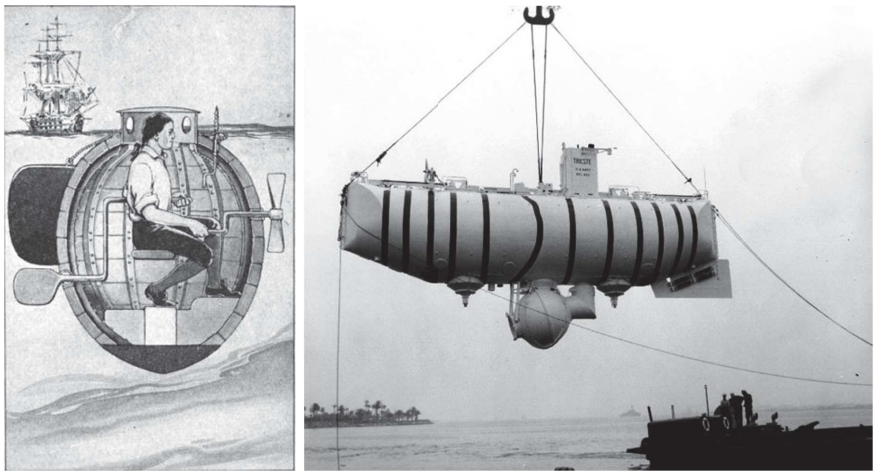

FIGURE 1.1 – Deux siècles séparent ces deux sous-marins : le *Turtle* [Bis16] à gauche et le *Trieste* à droite

En parallèle, les avancées sur les systèmes de plongée ainsi que le développement des caméras et appareils photos étanches accélèrent le succès de la plongée afin de découvrir les richesses du "monde du silence". Ainsi des années 40 à sa mort en 1997, Jacques-Yves Cousteau n'aura de cesse de parcourir les océans du globe afin de faire découvrir le milieu marin et sa fantastique richesse tout en nous révélant toute sa fragilité. Il sera d'ailleurs, avec ses équipes, à l'origine de nombreuses améliorations des techniques de plongée. Néanmoins, si l'Homme a été au centre de la conquête du monde sous-marin, il se révèle rapidement être le facteur limitant de son exploration. Dès 1841, le premier cas de décompression est répertorié. Les causes de celle-ci ne seront réellement comprises qu'à partir de 1878 et il faudra attendre 1908 pour que le principe des paliers de décompression soit proposé par Arthur Boycott, Guybon Damant et John Haldane.

Or les besoins d'exploration des milieux aquatiques sont de plus en plus grands afin de recueillir les données nécessaires à la compréhension de ces environnements, compréhension indispensable pour pouvoir exploiter au mieux leurs formidables ressources tout en préservant leur pérennité. Néanmoins les problèmes de sécurité nécessaires à l'envoi d'Hommes (plongeurs ou pilotes embarqués dans des sous-marins) et les coûts qu'ils engendrent dans de tels milieux limitent les explorations. Ainsi l'utilisation de robots permet d'évoluer dans ces environnements tout en supprimant les contraintes liées à la présence d'un opérateur humain, facilitant ainsi la réalisation de nombreuses applications. Mais en contrepartie les robots doivent soit permettre une téléopération sécurisée (en assurant l'évitement des obstacles par exemple), afin d'atteindre les objectifs de mission, soit pouvoir eux-mêmes embarquer l'expertise de l'opérateur humain

afin d'accomplir de manière autonome tout ou partie de la mission qui leur a été fixée.

1.1 Environnements aquatiques : enjeux, apports et défis de la robotique

Dans l'esprit du grand public, les robots sous-marins sont souvent considérés comme des engins imposants, tels que l'*Epaulard* [Ifr12] qui pèse plus de 20 tonnes, utilisés à des profondeurs importantes pour l'industrie pétrolière, des télécommunications ou pour l'étude des milieux océaniques. Si les applications sont nombreuses et importantes dans ce type d'environnements, nos travaux se situent dans le contexte applicatif des environnements faible fond (eaux terrestres, ports, littoral côtier) ou confinés (aquifères). En effet, dans ces milieux l'utilisation d'engins robotisés permet de répondre à de nombreuses problématiques scientifiques et commerciales. En outre, le faible volume navigable et la nécessité pour les utilisateurs de déployer une logistique légère imposent de recourir à des engins de faible encombrement et offrant une grande manœuvrabilité suivant les différents degrés de liberté du robot qui sont nécessaires à l'accomplissement des tâches à réaliser.

Nous allons présenter ici quelques exemples applicatifs illustrant les différents apports de la robotique dans ce contexte.

Les **barrages** sont utilisés afin de créer des retenues d'eau pour la production d'électricité ou l'irrigation, par exemple. Ils nécessitent un entretien fréquent afin d'éviter tout risque de rupture structurelle qui aurait des conséquences désastreuses. De fait, en France, les principaux barrages sont soumis tous les 10 ans à une inspection détaillée. Si auparavant cette inspection nécessitait une vidange complète du barrage afin de permettre une inspection minutieuse des parois, ce sont souvent des robots qui sont désormais utilisés afin de réaliser l'inspection [PBP00]. Cela nécessite que les robots puissent effectuer un suivi de paroi (c'est-à-dire de maintenir une orientation et une distance constante par rapport aux parois) tout en assurant un retour visuel satisfaisant pour permettre à l'opérateur de repérer tout dommage [MCC12]. Des caméras (possiblement complétées d'un plan laser pour améliorer l'évaluation des dimensions des anomalies) ou plus rarement un sonar à balayage sont utilisés pour observer la structure à inspecter et détecter les anomalies. Pouvoir suivre une trajectoire pour parcourir toute la surface du barrage et se positionner avec précision pour localiser plus facilement les anomalies sont également des sujets d'intérêt dans le cadre de cette application qui nécessitent l'utilisation d'une centrale inertielle (pour obtenir les orientations du robot) et d'un capteur de mesure de

vitesse comme un Loch Doppler [KRRH09].

Une autre problématique est celle de l'**envasement des barrages**. Celui-ci provient du fait que les barrages modifient la circulation des sédiments dans les cours d'eau. De fait ceux-ci s'accumulent autour du barrage, réduisant le débit de l'eau, dégradant sa qualité ou diminuant la durée de vie du barrage [McC01]. Les robots permettent de surveiller plus facilement l'évolution du niveau d'envasement en effectuant des mesures. Celles-ci permettent en outre de construire des modèles de l'évolution de l'envasement et du transport des sédiments qui aident ainsi à prédire l'évolution de l'envasement d'un barrage et à améliorer sa gestion. Des relevés de profondeur effectués à l'aide d'un sonar sédimentaire permettent d'estimer l'épaisseur de la couche de sédiments.

Les **canaux** sont des cours d'eau, parfois artificiels, utilisés pour l'irrigation, le transport marchand mais également pour la navigation de plaisance. Dans chaque cas, il est important d'entretenir régulièrement les canaux afin de maintenir leur navigabilité et l'état des berges du canal. Ceci permet soit de maintenir un débit important pour les canaux d'irrigation soit d'assurer une navigation sécurisée pour les bateaux. Or les canaux peuvent être comblés par des dépôts sédimentaires ou par des objets abandonnés dans le canal tandis que les berges dans les canaux de navigation peuvent être endommagées par le phénomène de batillage, c'est-à-dire par l'effet des vagues produites dans le sillage des bateaux à moteur. Des berges trop fragiles peuvent ensuite s'effondrer perturbant la navigation dans le canal. Mais l'entretien nécessite souvent de vider une portion de canal afin de vérifier son état puis d'entreprendre des opérations de nettoyage ou de réparation des berges si cela s'avère nécessaire [Clé09].

L'utilisation d'un robot permettrait donc d'assurer le suivi de l'état de celui-ci sans avoir besoin de le vider. Cela permettrait naturellement des gains importants en termes de coût et de temps et réduirait les perturbations du trafic liées aux interventions. Le robot doit être équipé d'un sonar profilométrique pour scanner le canal et en reconstruire le modèle afin qu'il soit étudié hors ligne pour décider des actions à entreprendre sur le canal. Des capacités comme le maintien dans l'axe du canal, le centrage dans celui-ci ou a contrario, le maintien d'une distance constante à l'une des berges, afin d'en réaliser un scan plus détaillé, sont des capacités qui permettraient de simplifier son utilisation dans le cadre de telles missions. Le maintien d'une profondeur constante par rapport au fond afin d'éviter tout risque d'envasement du robot est aussi une capacité pertinente. Pouvoir asservir le robot à une vitesse d'avance constante permet de simplifier la reconstruction du canal mais nécessite l'utilisation d'un capteur tel qu'un Loch Doppler. Enfin, il est nécessaire de mesurer l'évolution du cap du robot à l'aide d'une centrale inertielle afin de permettre là encore la reconstruction mais aussi pour pouvoir

asservir le cap du robot afin d'en simplifier le pilotage. La possibilité de naviguer en surface peut également permettre d'effectuer un recalage de la position du robot à l'aide d'un GPS afin d'améliorer la précision du modèle de canal reconstruit.

L'archéologie subaquatique regroupe l'étude des vestiges situés dans les eaux terrestres ou les zones côtières. Ces vestiges, qui peuvent être des épaves d'anciens navires ou d'anciens sites religieux, sont souvent préservés des dommages du temps par le milieu aquatique offrant une mine d'informations sur nos ancêtres et leurs habitudes de vie ou sur l'évolution du commerce fluvial à travers les âges. Il est important d'agir rapidement afin d'inventorier et protéger les différents sites archéologiques qui sont de plus en plus menacés par les activités humaines mais également convoités par de nombreux chasseurs de trésors. Le processus d'étude d'un site archéologique se déroule en plusieurs phases. Il est tout d'abord important de localiser avec précision le site de fouille. Il faut ensuite nettoyer celui-ci puis effectuer un relevé du site et notamment la position des différents artefacts, celle-ci représentant une source importante d'informations sur les habitudes de l'époque. Enfin, lorsque cela est possible, il faut extraire les objets du site et les répertorier.

Les robots ne sont pas encore réellement utilisables pour le nettoyage du site et l'extraction car c'est un travail nécessitant une minutie extrême et requérant toute l'expérience des archéologues afin de ne pas endommager les objets (soulignons néanmoins les travaux initiés par le projet *Corsaire* sur la manipulation dextre sous-marine [Car14]). Par contre, ils ont un réel intérêt pour la localisation et la cartographie des sites archéologiques. En effet, pouvant plonger pendant des durées bien plus grandes que les plongeurs, ils peuvent localiser bien plus rapidement les sites d'intérêts permettant de concentrer l'action des plongeurs sur les zones pertinentes. Les robots doivent pouvoir naviguer de manière autonome le long de la trajectoire qui leur a été imposée pour couvrir la zone d'étude tout en se localisant avec précision [ABC+14], [ACR+15]. Un sonar sédimentaire (i.e. qui permet de pénétrer les couches de sédiments) peut être utilisé pour effectuer le repérage d'objets enfouis. Pour la cartographie, l'utilisation d'un sonar à balayage couplé à une caméra permet une modélisation précise du site [CZSC09] et l'application de textures sur le modèle reconstruit afin d'en simplifier l'analyse.

Pour comprendre le fonctionnement complexe des milieux aquatiques (dans notre cas, les cours d'eau et les zones côtières où se concentrent les rejets d'eaux usées par exemple), il est nécessaire d'effectuer de nombreuses mesures. Ces mesures sont de plus en plus importantes avec les enjeux écologiques actuels pour comprendre l'impact du réchauffement climatique mais aussi évaluer l'impact engendré par certaines activités humaines notamment en termes

de pollution [DM12].

Les robots permettent donc d'effectuer ces **campagnes de mesures** plus efficacement et avec un coût opérationnel réduit [DM12]. Cela implique qu'ils aient une forte autonomie opérationnelle afin d'effectuer leurs mesures tout en évitant les obstacles [KUKW00]. Réaliser des transects [1] est aussi une fonctionnalité intéressante mais nécessite une navigation précise. Les robots doivent également pouvoir embarquer tous les capteurs constituant leur charge utile tels les différents sonars évoqués, ainsi que des capteurs de température, de salinité ou de profondeur (les trois pouvant être regroupés dans une sonde CTD). Des sondes physico-chimiques peuvent également s'avérer pertinentes dans le cadre de la recherche de polluants. Des fonctionnalités supplémentaires basées sur ces capteurs telles que la remontée de gradient sont également intéressantes car elles peuvent permettre de mieux comprendre l'origine d'un phénomène et de faciliter la localisation de sa source [GA02], [FPL05]. Enfin, la caractérisation du fond à l'aide d'un sonar bathymétrique et/ou d'un sonar à balayage est également utile [FZL+15].

De nombreuses espèces animales et végétales vivant dans le milieu aquatique constituent un indicateur important de son état de santé. En effet, elles sont sensibles à des variations de facteurs tels que la température ou la présence de polluants. Un suivi de l'évolution de leurs populations permet donc de comprendre les changements qui affectent le milieu aquatique et d'évaluer l'impact des activités humaines sur celui-ci [DM12]. En outre, la **protection de la biodiversité** passe par la réintroduction d'espèces dans des lieux (tels que les ports) où leur habitat a été dégradé à l'aide, par exemple, de la pose de récifs artificiels [2]. Le suivi des populations permet dans ce cas d'évaluer l'impact des mesures prises et d'entreprendre les actions correctives nécessaires. Cela nécessite donc d'effectuer des relevés et comptages réguliers. Mais leur réalisation par des humains est souvent longue et compliquée notamment à cause des limitations sur les durées des plongées.

Les robots, par leur capacité à opérer sur des durées bien plus longues, permettent donc de raccourcir ces campagnes d'évaluation [MWE07]. Ils doivent néanmoins être capables d'évoluer dans un environnement complexe en évitant les obstacles potentiels tels que les récifs ou les pontons. De fait, ils doivent être équipés d'une caméra pour permettre d'effectuer les relevés, et de capteurs proximétriques, tels qu'un sonar profilométrique ou un sonar sectoriel, afin de détecter les obstacles à proximité. Des capacités comme la tenue de position, d'orientation sont également pertinentes afin de faciliter le pilotage de l'engin et nécessitent donc centrale

1. Un transect est une référence spatiale virtuelle que le robot doit suivre afin d'étudier un phénomène
2. `http://www.ecocean.fr/`

inertielle et Loch Doppler. Des capteurs acoustiques, comme un échosondeur, peuvent également être mis en œuvre pour estimer les populations de végétaux présents sur le fond ainsi que leur répartition géographique [MLN+12]. Enfin, des possibilités d'asservissement visuel pour le suivi d'espèces ou l'observation de récifs constituent une fonctionnalité intéressante.

Les eaux souterraines sont l'une des principales sources d'eau douce dans le monde. En effet, si elles ne représentent qu'environ 0.68% des réserves d'eau totale de la planète [Pid06], cela correspond à un volume total d'environ 9.5 millions de kilomètres cube soit plus de 60 fois plus que les réserves présentes dans les lacs et rivières. Les **aquifères karstiques** constituent l'une des principales sources d'accès aux réserves situées à des faibles profondeurs et donc exploitables par l'Homme. Ils représentent donc une ressource importante en eau douce notamment dans des régions où l'eau est une ressource rare comme, par exemple, le Bassin Méditerranéen. Ces ressources sont à l'heure actuelle largement sous-exploitées [DJL+08] alors qu'elles représentent des enjeux économiques et sociétaux majeurs. Cela s'explique par la complexité de ces milieux. Les hydrogéologues doivent connaître assez précisément la géomorphologie de l'aquifère pour optimiser la gestion des ressources d'eau de celui-ci et éviter par exemple de dét"riorer (assèchement, remontée du biseau salin[3]) une source à cause d'un pompage incontrôlé. Or de par la complexité de leur structure, les aquifères karstiques ont un comportement hydrodynamique très spécifique [JML+15]. Il est donc nécessaire de posséder plus d'informations sur leur structure afin de mieux comprendre l'hydrodynamique des réseaux karstiques [Bak05]. Ces connaissances sont également nécessaires afin de pouvoir protéger à la fois les réseaux karstiques des activités humaines (engendrant de la pollution par exemple) et inversement prévenir les risques liés aux crues et inondations comme exposé dans [FLC+09], [NJP08] et [GPWJ14].

La problématique de la cartographie de ces aquifères et surtout des principaux conduits de leur réseau de drainage est donc essentielle dans l'exploitation et la gestion de la ressource en eau. Cela peut permettre de déterminer la meilleure zone où effectuer un forage afin de pomper avec un maximum d'efficacité l'eau d'un aquifère. Les méthodes issues de la géophysique telles que l'utilisation d'un géoradar ont une portée limitée, ne pouvant fournir que des informations partielles sur le réseau de conduits. Une autre approche pour déterminer les connexions entre les différents aquifères consiste à verser un traceur chimique dans l'un d'eux et d'évaluer la diffusion de ce traceur chimique à d'autres parties de l'aquifère. Néanmoins, cette méthode est limitée car elle ne donne pas d'information précise sur la structure même du conduit et

3. Un pompage mal contrôlé peut entrainer un déplacement de la zone d'interface entre l'eau douce et l'eau salée qui va alors venir "contaminer" la réserve d'eau la rendant inexploitable pour certaines applications telles que l'approvisionnement en eau potable.

certaines natures de parois, comme celles argileuses, peuvent absorber les traceurs. Une autre méthode couramment utilisée est le positionnement électromagnétique où un plongeur est amené à positionner un solénoïde ou un aimant dans le conduit [Bak05].

Cette dernière approche se révèle de plus en plus limitée par les interdictions de plongée dans les sources à cause des nombreux accidents mortels qui ont frappé les plongeurs lors d'explorations karstiques (voir l'exemple d'un accident de plongée dans Font Estramar en 2012 [Bou12]). Le remplacement des plongeurs par des robots s'avère donc une nécessité afin de poursuivre la cartographie et l'exploitation des aquifères, indispensables au vu de la croissance des besoins en eau potable des populations.

Néanmoins, faire évoluer des robots dans un environnement aussi complexe et peu structuré à des fins de cartographie s'avère un véritable défi scientifique et technologique. L'une des préoccupations prioritaires est de maintenir le robot à distance des parois en utilisant un sonar profilométrique qui permet également, tout comme une caméra acoustique, de reconstruire un modèle du réseau karstique qui constitue le livrable de mission. La seconde problématique est de faire naviguer le robot dans les conduits en gérant notamment les bifurcations du réseau. Pour cela il peut être nécessaire de se baser sur d'autres critères tels que les courants ou la température, nécessitant d'embarquer les capteurs pouvant mesurer ces grandeurs physiques, mesures également pertinentes lors de l'étude de l'hydrodynamique du réseau. Cela nécessite donc d'intégrer de la connaissance provenant des spécialistes et notamment des hydrogéologues. La communication avec l'opérateur est également très problématique car la présence d'un ombilical pour l'assurer est une option délicate à mettre en œuvre à cause du risque qu'il se coince ou soit sectionné par les parois des conduits. Il s'agit de la cause principale de la perte d'un robot dans ce type de mission [dFdV15] d'autant que sa récupération est souvent très complexe [Ber97] voire impossible. Cela impose que le robot ait une grande autonomie afin de pouvoir a minima atteindre un point de récupération seul si la communication par câble est perdue. Dans ce cadre, la reconstruction de l'environnement effectuée pour les hydrogéologues peut, potentiellement sous une forme dégradée (i.e. un modèle allégé en termes de quantités de données mais préservant les informations pertinentes à la navigation), servir à guider le robot dans le cadre d'une navigation autonome jusqu'au point de récupération. L'asservissement de la vitesse d'avance s'avère aussi nécessaire afin de simplifier le pilotage du robot, surtout en présence de courants, et nécessite l'utilisation d'un capteur tel qu'un Loch Doppler. Ce dernier permet en outre d'obtenir une mesure des courants dans lesquels navigue le robot ce qui ajoute des informations pertinentes pour l'étude du réseau.

L'exploration d'aquifères karstiques constitue notre objectif scientifique et applicatif. En

effet, la complexité de cet environnement pose de nombreux défis pour la conception de robots du point de vue matériel et logiciel. Il est notamment important de pouvoir s'appuyer sur l'expertise des spécialistes du milieu (hydrogéologues, plongeurs spéléos) afin de prendre les bonnes décisions pour explorer un conduit inconnu et pouvoir mener la mission de cartographie à terme sans risque pour le robot. Or, cette expertise est complexe à intégrer dans le logiciel de contrôle du robot car l'expérience et l'intuition de l'expert humain sont des atouts précieux face à un environnement qui demeure encore en grande partie inconnu. Cela nous a poussé à considérer le maintien d'un lien physique entre le robot et la surface, du moins pour le trajet aller, où l'expertise de l'opérateur humain doit nous guider dans l'exploration d'un environnement partiellement inconnu (par exemple, pour déterminer le conduit à suivre en cas de bifurcation).

Cependant, nous avons souligné précédemment les problèmes inhérents à l'ombilical. Pour les résoudre, nous avons envisagé de séparer les phases aller (exploration du réseau) et retour (au point de récupération) de la mission durant lesquelles le robot peut avoir des degrés d'autonomie plus ou moins importants suivant les besoins et le déroulement de la mission. Lors de la phase aller, le robot est relié par l'ombilical à la surface (qui peut en outre fournir de l'énergie au robot) et guidé par l'opérateur humain (il a de fait une autonomie décisionnelle très limitée) suivant son expertise tout en mettant en œuvre un certain nombre de fonctionnalités de sécurité (évitement des parois par exemple). Lors de la phase retour qui peut être soit décidée par l'opérateur soit causée par un problème dans la liaison au robot, ce dernier doit, en toute autonomie, pouvoir retourner au point de récupération en naviguant, par exemple, dans un modèle du réseau (simplifié) construit lors de la phase aller après s'être séparé du lien ombilical.

Cette autonomie variable a évidemment un impact sur la conception matérielle du robot. Afin de pouvoir se séparer du lien ombilical, il doit pouvoir emmener sa propre réserve d'énergie. De surcroit, il faut mettre en place un moyen pour le robot de détacher le câble. Cela pourrait passer par l'utilisation d'un trancanneur (un enrouleur) qui serait fixé au robot et pourrait être détaché par celui-ci en cas de besoin. Il permettrait en outre au robot de dérouler le câble réduisant les perturbations induites par celui-ci par rapport au cas où il serait déroulé depuis la surface et trainé par le robot.

Après cette brève description de divers exemples applicatifs, nous allons présenter les vecteurs robotiques pouvant être mis en œuvre dans ces applications.

1.2 Les différents types de robots sous-marins

Comme nous l'avons vu, les robots sous-marins, de par leur capacité à intervenir là où il est dangereux d'envoyer des plongeurs ou des submersibles habités, ont de grandes perspectives en termes d'applications.

On peut distinguer deux grandes familles de robots sous-marins : les Remotely Operated Vehicles (ROVs) et les Autonomous Underwater Vehicles (AUVs). Nous allons présenter les spécificités de chacun d'entre eux et quelques défis techniques et scientifiques associés à leur mise en œuvre.

1.2.1 Les ROVs

Les ROVs sont reliés physiquement à la surface via un lien ombilical. Ce dernier permet la communication avec l'opérateur de surface et assure le plus souvent l'alimentation du robot en énergie depuis la surface, simplifiant sa conception. Ils sont souvent téléopérés, c'est-à-dire que l'opérateur en surface envoie directement les consignes à appliquer aux actionneurs en fonction du retour des différents capteurs dont est équipé le robot. Ils furent un sujet de recherche très important dans les années 60 et 70 et commencèrent à être régulièrement utilisés dans l'industrie à partir des années 80.

Cette technologie est donc relativement mature et de nombreux ROVs sont déjà en service dans l'industrie. En effet, la très grande majorité des engins qui interviennent lors d'opérations de maintenance d'infrastructures immergées (industrie pétrolière, télécommunications ou barrages) ou pour l'inspection d'épaves sont des ROVs. Ils sont en outre souvent équipés d'un bras manipulateur qui permet d'effectuer une action sur l'environnement du robot. On peut ainsi citer le ROV Spectrum [Oce12] (140cm x 90cm x 85cm, 290kg) ou le SeaEye Cougar XT [Tam13] (151cm x 100cm x 79cm, 400kg). De par la nature de leurs missions, qui nécessitent une grande manœuvrabilité et donc des mouvements suivant tous les degrés de liberté sans direction préférentielle, ils sont iso-actionnés. Les degrés de liberté non contrôlés sont naturellement stabilisés par la structure de l'engin. De fait, cette manœuvrabilité facilite leur intervention dans un faible volume d'eau et dans un milieu potentiellement encombré. On peut par exemple citer le ROV Super Achille conçu par la COMEX [COM15] (72cm x 68cm x 60cm ,110kg), le DTX2 conçu par Deep Trekker [Tre15] (63cm x 49cm x 38cm, 26kg), le Phantom T3 conçu par Deep Ocean [Oce15] (91cm x 39cm x 35cm, 36kg), le Pro 4 conçu par VideoRay [Vid14] (37cm x 29cm x 22cm, 6kg) ou le AC-ROV 100 conçu par AC-CESS [AC14] (20cm x 15cm x 14cm, 3kg).

Néanmoins, lorsqu'ils sont soumis aux contraintes du milieu aquatique (notamment les

courants), leur pilotage s'avère délicat. Ce phénomène est accentué par la présence du lien ombilical qui génère d'importantes perturbations. C'est pour cela que les pilotes de ROVs doivent souvent avoir suivi de nombreuses heures de formation et de mise en pratique avant d'être autorisés à les piloter lors d'une intervention.

Ainsi les travaux se concentrent essentiellement sur un accroissement de l'autonomie opérationnelle de ces engins afin d'en simplifier le pilotage pour que l'opérateur puisse se concentrer sur la réalisation des tâches qui nécessitent le plus de précision et où l'expertise humaine s'avère indispensable. De plus un pilotage plus simple ouvrirait l'utilisation des ROVs à un public plus varié et notamment les scientifiques qui n'auraient alors plus besoin de recourir à des intervenants extérieurs pour réaliser leurs campagnes de mesure. De plus en plus d'asservissements sont donc embarqués sur ces robots, que ce soit un maintien de cap, un asservissement en profondeur ou la capacité à maintenir leurs position et orientation spatiales. Mais des travaux orientent les recherches sur ces robots vers encore plus d'autonomie opérationnelle, pour ne laisser à l'opérateur que la prise de décision. Cela inclut par exemple la possibilité de se rendre automatiquement au lieu de mission ou encore d'éviter les obstacles qui se présentent sans nécessiter l'intervention de l'opérateur. Qui plus est, afin d'accroître leur autonomie et de réduire leur dépendance au lien ombilical, certains ROVs embarquent leur propre réserve d'énergie. Cela permet en outre d'utiliser un câble plus fin pour transmettre les données réduisant son impact sur le comportement du robot. Dans ce cadre, on peut constater que les recherches tendent à rapprocher les ROVs du fonctionnement des AUVs, la principale différence étant le maintien d'une présence forte de l'opérateur dans le processus de commande notamment lorsqu'il s'agit de prendre des décisions sur la conduite de la mission.

Néanmoins l'une des principales limitations des ROVs est l'ombilical. En effet, outre les perturbations qu'il génère et qui pénalisent le comportement de l'engin et doivent donc être compensées, il limite le champ d'action des ROVs puisque le robot ne peut opérer sans contact avec l'opérateur si le câble est endommagé.

1.2.2 Les AUVs

La seconde catégorie, les AUVs (Autonomous Underwater Vehicles) aussi appelés UUVs (Unmanned Underwater Vehicles), regroupe les robots autonomes. Bien que proposant de nouveaux défis scientifiques et technologiques qui demeurent encore, en grande partie, à solutionner, ils offrent des perspectives importantes en termes d'applications réalisables. En effet, détachés des contraintes de liaison à la surface, ils peuvent effectuer des missions sur des durées et des distances bien plus importantes tout en pouvant se rendre dans des lieux où la présence d'un ombilical est problématique. Leur développement commença à partir des années 70 et

au début des années 80. Depuis lors les recherches n'ont cessé afin d'améliorer les AUVs pour qu'ils puissent réaliser des missions de plus en plus complexes avec une robustesse accrue. Les AUVs ont longtemps été majoritairement des robots de taille importante et sous-actionnés (par exemple les torpilles) destinés à évoluer dans les vastes étendues océaniques. On peut ainsi citer le REMUS 600 [Ins10] une torpille de 32cm de diamètre pesant 240kg, le HUGIN 1000 [Kon15] (75cm de diamètre, 450cm de long, 650kg), le Bluefin-9 de Bluefin Robotics [Rob10] (24cm de diamètre, 175cm de long, 60kg), l'AUV Taipan [JAJ05] (25cm de diamètre, 190cm de long, 40kg) ou plus récemment l'AUV MARTA [ABC+14] (18cm de diamètre, 300cm de long, 70kg).

Néanmoins, les besoins d'exploration d'environnements confinés ou faible fond, dans lesquels l'encombrement du robot doit être très limité et sa manœuvrabilité importante, poussent vers le développement d'AUVs de petite taille et iso-actionnés, d'une structure d'actionnement proche de celle des ROVs. On peut ainsi citer l'AUV ODIN [ACSW01], l'AUV Nessie [MCJP10] (70cm x 70cm x 60cm, 41kg) ou encore un AUV bioinspiré, le U-CAT [ACR+15].

En outre, ce rapprochement comporte également le désir de faire intervenir les AUVs de manière autonome lors d'interventions de réparation (autrefois chasse gardée des ROVs) et a donné naissance à la classe des *Intervention AUVs* (I-AUVs) [PRP+12]. Ceux-ci sont des AUVs équipés d'un bras manipulateur. Cela ouvre de nouvelles perspectives en termes d'applications avec la possibilité de réaliser de nouvelles actions telles que le prélèvement d'échantillons, la réparation ou la récupération de capteurs déployés dans l'environnement. Bien que ces missions soient tout à fait complémentaires avec celles de collecte de données dans le cadre de l'étude des milieux aquatiques, nous ne nous focaliserons que sur l'utilisation des robots sous-marins dans le contexte de l'acquisition de données scientifiques.

Cela pose de nouveaux défis scientifiques notamment au niveau de l'autonomie de ces engins. De fait, il faut concevoir le contrôle du robot afin que celui-ci puisse accomplir sa mission avec la réserve d'énergie dont il dispose. Par ailleurs, l'absence de lien de communication de qualité sous l'eau, empêchant des échanges de données réguliers et importants avec un opérateur, oblige à concevoir le logiciel de contrôle du robot pour qu'il puisse mener à bien une mission de manière entièrement autonome. Cela impose notamment de donner au robot les moyens d'atteindre ses objectifs de mission et implique qu'il puisse embarquer l'expertise de l'opérateur humain tant pour la réalisation des tâches que pour la mise en place des processus décisionnels. Les problématiques liées à ce fonctionnement autonome et la limitation des possibilités d'intervention d'un opérateur en cas de problème posent de nombreux défis scientifiques :

– Modélisation hydrodynamique et prise en compte de l'étage d'actionnement : les effets

de l'interaction d'un robot sous-marin avec le milieu aqueux ne peuvent être négligés car ils influent fortement sur le comportement du robot lorsqu'il se déplace même à faible vitesse. Il est donc nécessaire de modéliser les interactions hydrodynamiques entre l'AUV et l'eau [AFY08]. Celles-ci dépendent de la structure du robot (forme, symétries, masse) et peuvent lui donner des directions préférentielles de mouvement. Mais la nature du milieu marin joue également car des paramètres tels que la salinité ou la température modifient également les effets hydrodynamiques. Les actionneurs doivent également être pris en compte. En effet des différences, même faibles, entre les caractéristiques (i.e. la force produite en fonction de la consigne appliquée) des actionneurs sont accentuées par le milieu marin et perturbent le comportement du robot. Il est donc nécessaire de prendre en compte et de corriger ces écarts.

– Navigation : Les ondes électromagnétiques étant fortement atténuées sous l'eau, certains systèmes de localisation comme le GPS deviennent inutilisables. Il est donc nécessaire de mettre en place d'autres solutions afin d'estimer la position et la vitesse du robot. Si cette dernière peut être estimée en utilisant un Loch [4] (notamment les Loch Doppler [5]), la nécessité d'intégrer ces vitesses entraine une dérive qu'il est nécessaire de réajuster régulièrement. D'autres stratégies doivent donc être mises en place, il peut s'agir de remonter pour réaliser des points GPS lorsque la dérive de position devient trop importante mais cette solution ne peut être utilisée dans environnements confinés comme les conduits karstiques. Il est également possible d'instrumenter le milieu en positionnant par exemple des amers. Néanmoins, cette stratégie est limitée car elle nécessite d'avoir accès au milieu préalablement ce qui est à la fois impossible dans certains environnements dont l'accès est dangereux mais aussi contradictoire avec la volonté d'utiliser ces robots avec une logistique aussi légère que possible. Enfin, lorsque des connaissances préalables existent sur l'environnement (une carte par exemple), celles-ci peuvent aider à la localisation du robot.
– Intégration de l'expertise humaine : Puisque l'AUV doit fonctionner de manière totalement autonome, il est nécessaire d'intégrer les processus et critères décisionnels qui sont réalisés par l'opérateur. Cela inclut les modèles de l'environnement qui permettent au robot de prendre en compte celui-ci afin de réaliser sa mission au mieux et les critères décisionnels qui permettent de déterminer si les objectifs de mission sont atteints ou

4. Un Loch est un capteur permettant de mesurer la vitesse d'un navire ou d'un sous-marin soit par rapport au fond soit par rapport à la masse d'eau dans laquelle il évolue.

5. Les Loch Doppler mesurent la vitesse en utilisant l'effet Doppler, c'est-à-dire le décalage de fréquence d'une onde, typiquement acoustique, observée lorsque la distance entre émetteur et récepteur varie au cours du temps.

doivent être réajustés en fonction de la situation. Mais comme ces modèles proviennent souvent d'autres disciplines scientifiques telles que l'hydrogéologie ou la biologie, les automaticiens ne sont pas forcément familiers avec ces concepts. Leur intégration aux lois de commande nécessite que ces connaissances soient suffisamment explicites pour être manipulées simplement.

- Adaptation des objectifs : De nombreux évènements "inattendus" peuvent se produire et il est nécessaire d'y réagir rapidement afin de maintenir l'intégrité du robot et de permettre autant que possible la poursuite de la mission. Cela implique aussi de pouvoir intégrer et représenter les conditions qui dictent les changements de stratégie de commande et les connaissances qui leur sont associées.
- Communications : Même si les robots sont capables de fonctionner de manière autonome, maintenir un lien de communication avec la surface est important. Cela permet à l'opérateur de suivre le déroulement de la mission, d'ajuster les objectifs si nécessaire, d'être informé d'un éventuel problème rencontré par l'AUV et de récupérer un certain nombre de données au fur et à mesure afin de limiter les dommages causés par la perte d'un robot. Or les communications sous-marines, basées sur des ondes sonores, sont fortement limitées dans le rapport entre débit et portée. D'autres stratégies telles que la mise en place d'un ombilical détachable si jamais il venait à menacer l'intégrité du robot, en se coinçant par exemple, sont donc à considérer.
- Gestion de l'énergie : Les AUVs devant embarquer leur propre réserve en énergie, il est nécessaire de mettre en place les mécanismes permettant de gérer celle-ci d'une manière efficace afin d'optimiser l'utilisation de cette ressource limitée. Cela impose de choisir les stratégies de commande les plus sobres du point de vue énergétique ou de n'utiliser certains systèmes (comme les capteurs) que lorsque cela est nécessaire.

Toutes ces problématiques doivent être prises en compte lors de la description du contrôleur du robot et ensuite répercutées lors de l'implémentation de celui-ci sur l'architecture logicielle de contrôle du robot. Dans le contexte de nos travaux, les problématiques d'énergie et de communication ne seront pas abordées tout comme l'adaptation des objectifs de mission.

Nous allons maintenant présenter le robot utilisé pour mener à bien nos expérimentations.

1.3 Entre ROV et AUV : Le Jack

Avant d'introduire le robot que nous mettons en œuvre, nous allons rappeler les notations utilisées en robotique sous-marine.

1.3.1 Notations

La Figure 1.2 présente les deux principaux repères utilisés en robotique sous-marine. $\{U\} = \{\vec{x}_u, \vec{y}_u, \vec{z}_u\}$ est le repère monde qui sert de référence. L'axe $\vec{x}_u$ est orienté vers le Nord, $\vec{y}_u$ vers l'Est et $\vec{z}_u$ vers le fond (ainsi $\{U\}$ est un repère direct). Le repère $\{B\} = \{\vec{x}_B, \vec{y}_B, \vec{z}_B\}$ est le repère robot [Lap06].

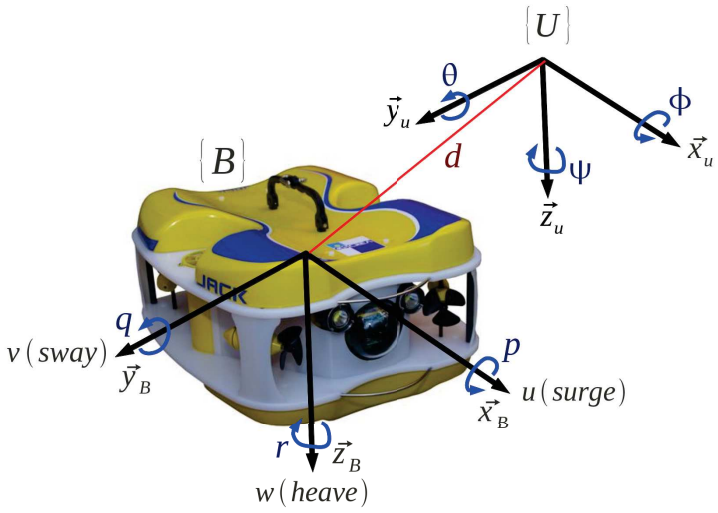

FIGURE 1.2 – Le repère robot $\{B\}$ et le repère monde $\{U\}$

$[\phi, \theta, \psi]$ désignent les orientations du robot dans le repère monde, respectivement le roulis, le tangage et le lacet (cap).

$d = [x, y, z]$ représente la position de $\{B\}$ par rapport à $\{U\}$.

$[u, v, w]$ désignent les vitesses linéaires du robot exprimées dans $\{B\}$. Il s'agit respectivement des vitesses d'avance (*surge*), de glissement (*sway*) et de déplacement vertical (*heave*).

$[p, q, r]$ sont les vitesses angulaires autour des axes $\vec{x}_B$, $\vec{y}_B$ et $\vec{z}_B$ respectivement.

On notera $F_B = [F_u, F_v, F_w, \Gamma_p, \Gamma_q, \Gamma_r]^\intercal$ le vecteur des forces appliqué au robot dans le cadre de la commande.

1.3.2 Le Jack

Le robot que nous utiliserons pour illustrer nos travaux sera le ROV Jack (Figure 1.3) développé à partir d'une version commercialisée par la société Ciscrea. Dans sa configuration de base, ce ROV mesure 54 cm de long, 41 cm de large et 26 cm de haut tout en ne pesant que 11 kg hors de l'eau (hors éventuelle charge utile). Il est donc parfaitement adapté à notre contexte applicatif puisqu'il peut facilement naviguer dans un volume navigable restreint et ne nécessite pas une logistique lourde pour le transporter et le déployer.

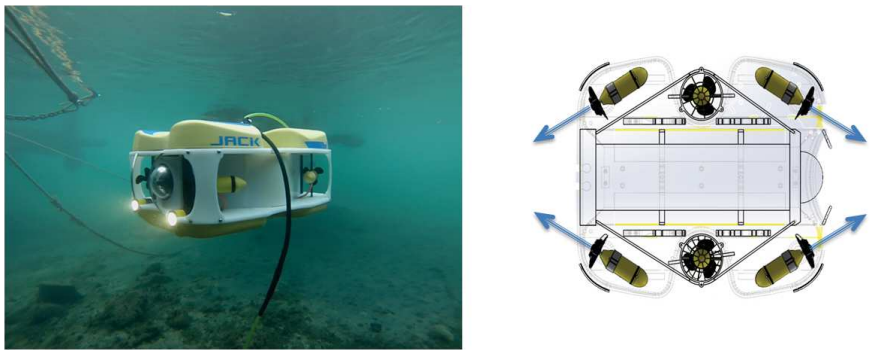

FIGURE 1.3 – Le ROV *Jack* et son actionnement

De plus la structure vectorielle de ses moteurs lui assure une grande manœuvrabilité indispensable dès lors que le robot doit évoluer dans un faible volume d'eau et un environnement potentiellement encombré. Quatre moteurs horizontaux permettent des déplacements suivant l'avance (u), le glissement (v) et en rotation autour de l'axe $\vec{z}_B$ (r). Deux moteurs orientés verticalement permettent de faire évoluer sa profondeur (w) et de contrôler son roulis (p). Le tangage (q) n'est pas contrôlable mais est naturellement stabilisé.

Le Jack transporte également deux batteries qui assurent son autonomie énergétique. L'échange de données avec le robot est réalisé à l'aide d'un ombilical d'une longueur de 80 mètres permettant une liaison Ethernet avec l'engin. De par le fait que le Jack embarque sa propre réserve d'énergie, il est envisageable de se passer de ce câble et de faire fonctionner ce robot en mode AUV avec une autonomie décisionnelle et opérationnelle élevée. Le fonctionnement en mode AUV peut être utilisé soit parce que l'application le permet car elle ne nécessite pas une prise de décision par l'opérateur soit parce que les circonstances de mission l'imposent car l'ombilical peut s'être coincé entre des rochers par exemple. Cette évolution est d'autant plus importante à envisager que les perturbations importantes induites par l'ombilical rendent très délicates la téléopération ou la tenue des asservissements.

Le contrôleur matériel du robot est une BeagleBone Black Révision C [Col14]. Cette carte assure la gestion de l'électronique du robot, permet de réaliser les asservissements de base (cap ou profondeur, par exemple) et assure le transfert des données vers la surface. Elle est dotée d'un noyau Linux patché avec Xenomai 2.6 sur lequel est installé le *Middleware* temps-réel ContrACT. Ce dernier sera présenté plus précisément au Chapitre 7.

Le Jack est équipé de plusieurs capteurs. Une caméra IP permet la remontée d'images via le lien Ethernet. Le robot dispose également d'un capteur de profondeur et d'une centrale inertielle (XSens MTI) nous permettant d'obtenir les orientations (roulis, tangage et cap), les vitesses angulaires dans le repère robot (p, q et r) ainsi qu'une estimation des accélérations linéaires subies par le robot dans $\{B\}$ ($\dot{u}$, $\dot{v}$ et $\dot{w}$). D'autres capteurs fournissent la température

interne du robot, la température de l'eau, l'état de charge des batteries et la détection de voies d'eau.

Néanmoins, la taille réduite du Jack, si elle permet la navigation dans les milieux visés, limite également la charge utile embarquable. Dès lors, il est nécessaire de cibler les capteurs à utiliser pour chaque application visée afin de n'embarquer que ceux qui sont pertinents. Il faut alors que l'architecture matérielle et la structure mécanique de l'engin soient modulaires afin de faciliter l'adaptation de la charge utile du robot aux spécificités de chaque mission. L'architecture matérielle présentée ci-dessus intègre donc ces préoccupations grâce à deux prises Ethernet, permettant l'ajout de capteurs de charge utile, de puissance de calcul supplémentaire (carte de contrôle haut-niveau), et autorisant également l'ajout d'un étage d'actionnement supplémentaire.

Du point de vue mécanique, un *skid*, comme celui présenté aux Figures 1.4 et 1.5, permet de fixer différents capteurs au robot. Lorsque la capacité d'emport du robot devient insuffisante, il est également possible de mettre en œuvre une version comprenant un second étage d'actionnement relié mécaniquement à l'étage supérieur, comme illustré à la Figure 1.6. Cette version sera présentée de manière plus détaillée au Chapitre 8.

FIGURE 1.4 – Le ROV Jack équipé d'un sonar sectoriel monté sur un *skid*

Il apparaît dès lors que, pour pouvoir tirer parti de la modularité matérielle du Jack, le logiciel assurant son contrôle doit s'adapter aux différentes versions du robot. Outre la possibilité d'accéder aux données fournies par les capteurs en proposant les *drivers* adéquats, il doit également permettre d'embarquer, suivant les besoins de chaque application, les connaissances qui permettent d'utiliser les informations fournies par les capteurs pour le contrôle du robot. Cela souligne que le contrôleur du robot doit lui aussi être modulaire pour exploiter au mieux la polyvalence offerte par l'architecture matérielle.

FIGURE 1.5 – Le ROV Jack équipé d'un sonar profilométrique monté sur son *skid*

1.4 Points clés du chapitre

- Les applications de la robotique sous-marine dans les environnements confinés ou faible fond sont nombreuses et diverses.
- Ces environnements nécessitent l'utilisation de robots de faibles dimensions.
- Les robots sous-marins se dirigent vers une autonomie opérationnelle toujours plus importante.
- L'interaction entre un robot sous-marin et le milieu aquatique étant forte, cette autonomie passe par l'intégration de modèles de l'environnement au contrôle.
- Le développement d'un robot et de son contrôleur est donc un processus interdisciplinaire faisant intervenir des connaissances très variées.
- Il est nécessaire d'adapter les robots et leur contrôle aux spécificités de chaque application.
- De nombreuses problématiques de contrôle sont toutefois transversales entre les différentes applications.

FIGURE 1.6 – Une version 12 moteurs permet l'emport simultané de plusieurs capteurs

Chapitre 2

Contrôle de robots : modularité et stabilité

Comme nous l'avons vu au chapitre précédent, les recherches en robotique sous-marine se dirigent vers un accroissement de l'autonomie opérationnelle des engins. Cela nécessite de développer les lois de commande décrivant le contrôleur des robots puis de les implémenter sous forme de logiciel de contrôle. Ce dernier est structuré suivant une architecture logicielle.

Les besoins applicatifs sont nombreux. S'ils partagent certaines problématiques de commande, d'autres modèles ou problématiques sont spécifiques à chaque application. Afin de répondre aux besoins, il est donc nécessaire de gagner en efficacité dans le développement des commandes depuis leur conception et jusqu'à leur mise en œuvre. De fait, une conception modulaire des lois de commande devient une nécessité pour répondre aux défis applicatifs de la robotique et atteindre les propriétés d'évolutivité et de réutilisabilité indispensables pour un développement efficace.

Cette problématique est depuis longtemps au cœur des recherches en informatique. Ainsi les différentes approches ont donné naissance à diverses architectures logicielles qui visent à favoriser la réutilisabilité des différents composants logiciels et à permettre plus de flexibilité dans leur développement. A contrario, ces problématiques ont longtemps été laissées de côté par les automaticiens lors de la conception des lois de commande. Ces dernières sont dès lors souvent monolithiques, mélangeant les connaissances relevant de différents domaines et les caractéristiques de la cible d'implémentation, ce qui rend très difficile leur réutilisation dans un contexte applicatif différent. Néanmoins, suite à une prise de conscience progressive, des approches ont commencé à tenir compte de la problématique de modularité. Il est en outre nécessaire de concevoir les commandes avec un niveau d'abstraction suffisant pour que leur description demeure aussi indépendante que possible de la cible d'implémentation.

De plus, la manière d'implémenter une loi de commande va grandement influencer le comportement du robot et sa stabilité. Il faut donc pouvoir exprimer les contraintes émanant de l'implémentation et les mettre en relation avec celles provenant des besoins de l'automaticien, comme la stabilité. De fait les interactions doivent être fortes entre les automaticiens, qui conçoivent les lois de commande, et les informaticiens qui les mettent en œuvre. Or ces interactions sont insuffisantes principalement car ces deux domaines travaillent avec des préoccupations exprimées dans des espaces différents et ne trouvent pas de terrain commun de représentation. Il est donc important de définir une base permettant ces échanges autour de la mise en commun des besoins et contraintes provenant des deux domaines.

2.1 Stabilité d'un contrôleur

Nous allons commencer par rappeler les outils théoriques permettant d'étudier la stabilité d'un contrôleur. Nous présenterons les outils couramment utilisés afin de démontrer la stabilité d'un contrôleur en temps continu. Néanmoins, l'exécution d'un contrôleur étant réalisée de manière discrète sur un calculateur, nous expliquerons comment la notion de stabilité doit alors être enrichie pour prendre en compte cette réalité et mettrons en lumière les outils associés.

2.1.1 Systèmes linéaires

Commençons d'abord par illustrer la problématique de stabilité dans le cadre des systèmes linéaires où les outils théoriques sont bien établis à la fois en temps continu et en temps discret.

Dans le cas où le système asservi est linéaire, l'étude de sa stabilité se base souvent sur la détermination de sa fonction de transfert qui se fait généralement en utilisant la transformée de Laplace. Soit le système représenté dans la Figure 2.1, où $C(p)$ est la fonction de transfert du contrôleur, $H(p)$ celle du système et $G(p)$ celle des capteurs.

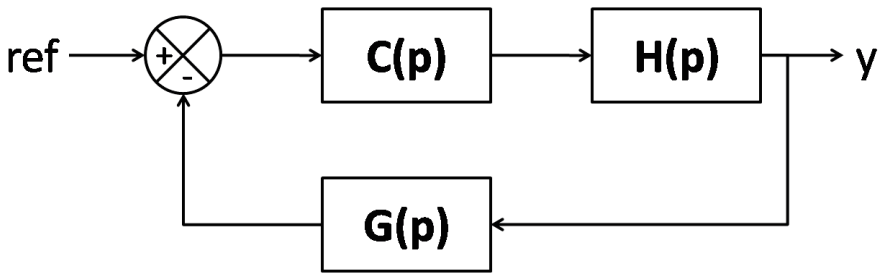

FIGURE 2.1 – Structure d'un système asservi

La fonction de transfert du système, $H_{BF}(p)$ est obtenue par la formule :

$$H_{BF}(p) = \frac{H(p)C(p)}{1 + H(p)C(p)G(p)} \tag{2.1}$$

Une fois obtenue, on peut déterminer la stabilité avec la règle suivante :

Un système linéaire est asymptotiquement stable si et seulement si les pôles de sa fonction de transfert sont à partie réelle strictement négative.

Exemple 2.1:

Illustrons cette approche sur un exemple trivial. Considérons le contrôleur présenté à la Figure 2.2. Le système à contrôler est décrit par une équation différentielle d'ordre 2 avec un retour unitaire et un contrôleur proportionnel.

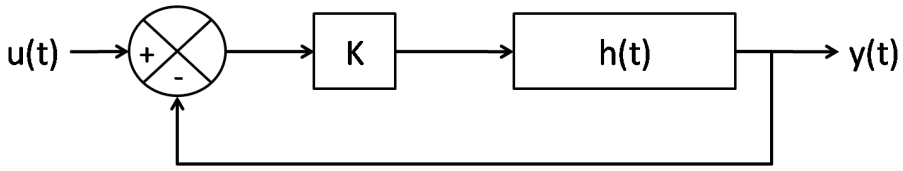

FIGURE 2.2 – Un exemple simple de système linéaire

En transformée de Laplace, nous pouvons écrire :

$$H(p) = \frac{K_s}{(p - p_1)(p - p_2)} \tag{2.2}$$

$$G(p) = 1 \tag{2.3}$$

$$C(p) = K \tag{2.4}$$

On peut donc écrire par l'équation (2.1) :

$$H_{BF}(p) = \frac{KK_s}{p^2 + p(-p_1 - p_2) + p_1p_2 + KK_s} \tag{2.5}$$

A partir de cette équation, pour un système donné et connu, on peut vérifier si une valeur de K rendra le système contrôlé stable ou non. Par exemple si l'on pose $K_s = 2$, $p_1 = -0.1 + 1.5j$, $p_2 = -0.1 - 1.5j$ et $K = 1$, les pôles de la fonction de transfert sont $-0.1 - 2.0616j$ et $-0.1 + 2.0616j$. Leurs parties réelles étant négatives, le système contrôlé est stable.

Or, nous avons précédemment souligné que l'exécution du contrôle sur un calculateur s'effectuait en temps discret. Le choix des périodes d'exécution a dès lors un impact majeur sur la stabilité du système contrôlé. Il est donc nécessaire de s'intéresser à l'étude de la stabilité en temps disret.

Dans le cas des systèmes linéaires, il est tout d'abord nécessaire d'introduire dans le système des bloqueurs d'ordre 0 qui représentent les échantillonnages réalisés dans le système de par l'exécution en temps discret. Ensuite il faut calculer la transformée en z de la fonction de transfert. Le système est alors stable si et seulement si :

Les pôles de sa fonction de transfert se situent à l'intérieur du cercle unité (i. e. leur norme est inférieure à 1).

Exemple 2.2:

Reprenons l'exemple précédent.

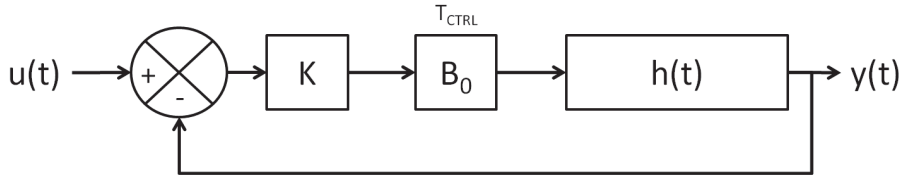

FIGURE 2.3 – L'ajout d'un bloqueur d'ordre 0 permet de matérialiser la période du contrôleur

$$KB_0(z)H(z) = K\frac{A_1 z + A_0}{(z - z_1)(z - z_2)} \tag{2.6}$$

avec

$$z_1 = e^{p_1 T_{CTRL}}$$
$$z_1 = e^{p_2 T_{CTRL}}$$

On obtient donc

$$H_{BF}(z) = \frac{Num(z)}{(z - z_1^{BF})(z - z_2^{BF})} \tag{2.7}$$

A partir de ce point, les calculs, déjà complexes malgré la simplicité du système, seront réalisés à l'aide de Matlab. Pour que le système soit stable, il faut que $\|z_{1,2}^{BF}\| \leq 1$. Or $z_{1,2}^{BF} = f(K, T_{CTRL})$. Deux options s'offrent donc pour étudier la stabilité d'un tel système.

Premièrement, pour une valeur de K donnée, il est possible de déterminer l'ensemble des valeurs de T_{CTRL} pour lesquelles le système reste stable.

Nous reprenons les mêmes valeurs numériques que pour l'exemple précédent : $K_s = 2$, $p_1 = -0.1 + 1.5j$, $p_2 = -0.1 - 1.5j$ et $K = 1$. La Figure 2.4 illustre la valeur maximale de la norme des pôles.

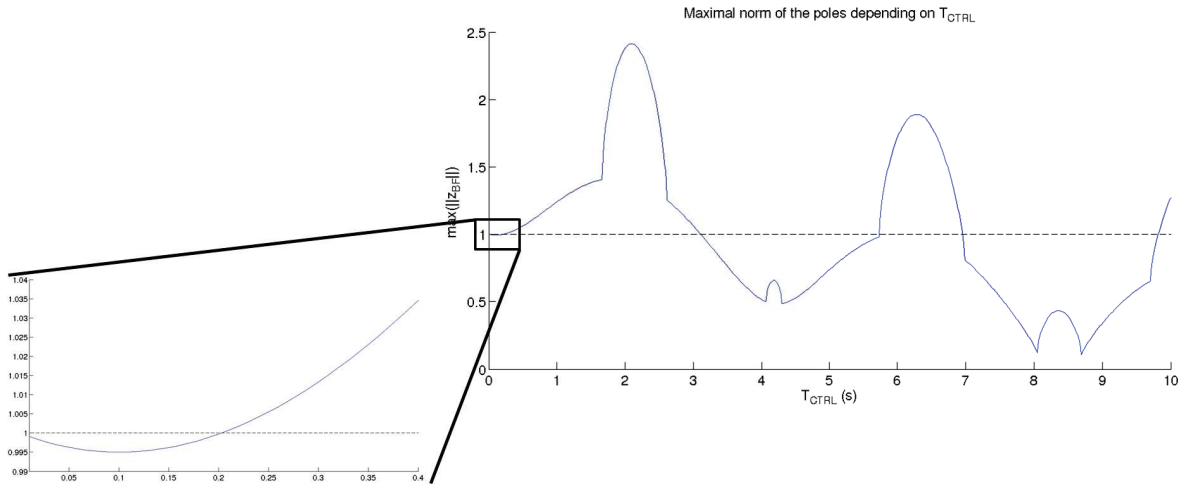

FIGURE 2.4 – Norme maximale des pôles en fonction de T_{CTRL}. Si cette norme est inférieure à 1, le système est stable

Dans ce cas, on voit que le contrôle est stable si $T_{CTRL} \in [0\ 0.2] \cup [3.11\ 5.73] \cup [6.96\ 9.81]$.

Deuxièmement, pour une valeur de T_{CTRL} donnée, il est possible de déterminer l'ensemble des valeurs de K pour lesquelles le système reste stable.

Nous reprenons les mêmes valeurs numériques du système que pour l'exemple précédent : $K_s = 2$, $p_1 = -0.1 + 1.5j$, $p_2 = -0.1 - 1.5j$ et choisissons $T_{CTRL} = 100ms$. La Figure 2.5 illustre la valeur maximale de la norme des pôles.

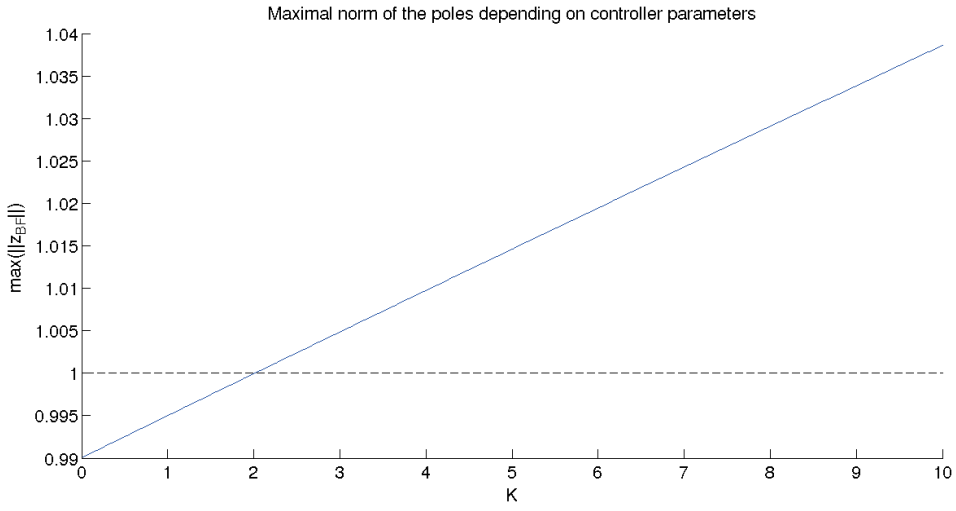

FIGURE 2.5 – Norme maximale des pôles en fonction de K. Si cette norme est inférieure à 1, le système est stable

Dans ce cas, on voit que le contrôle est stable si $K \in [0.01\ 2]$.

Ces deux approches sont bien entendu complémentaires. En effet, il est, par exemple, possible de commencer par déterminer l'ensemble des périodes d'exécution possibles selon le paramétrage choisi du contrôleur. Puis, une fois la période d'exécution choisie parmi l'ensemble des possibles, il est intéressant de déterminer la marge de manœuvre possible pour modifier le paramétrage du contrôleur. Une autre option peut être de faire évoluer non pas les paramètres du contrôleur mais ceux du système contrôlé. Dès lors, il est possible de connaître l'ensemble de leurs valeurs pour lesquelles le système contrôlé restera stable à la période choisie. Cela permettra d'estimer à quel point des erreurs de modélisation du système peuvent impacter la stabilité de celui-ci. Ainsi, il est possible de, par exemple, choisir la période d'exécution en fonction de la marge de manœuvre qu'elle laisse vis-à-vis des erreurs de modélisation.

A travers cet exemple simple, il apparait clairement que la notion de **stabilité** doit être, en temps discret, augmentée d'un **ensemble de périodes**. De fait, l'implémentation d'une loi de commande passe nécessairement par le choix de la période à laquelle celle-ci devra s'exécuter sur le calculateur. Dès lors, nous voyons que cette notion de période d'exécution doit devenir un élément central de l'étude de la stabilité des équations de contrôle.

2.1.2 Systèmes non linéaires

Dans le cadre des systèmes non linéaires, il n'est pas possible d'obtenir l'expression de la fonction de transfert. Or la plupart des systèmes robotiques contiennent des non linéarités. Il faut donc également s'intéresser aux outils théoriques qui permettent d'en étudier la stabilité.

La théorie de Lyapunov, présentée dans [SL91], est principalement utilisée. Celle-ci comprend deux méthodes. La méthode indirecte se base sur le fait que la stabilité d'un système non linéaire autour d'un point d'équilibre peut être approchée par celle de son approximation linéaire. Elle permet ainsi d'appliquer les techniques des systèmes linéaires pour déduire la stabilité d'un système non linéaire. Néanmoins cette approche étant restreinte au cas des points d'équilibres, son application est relativement limitée.

La méthode directe de Lyapunov est donc la plus communément utilisée dans la littérature. Elle se base sur la théorie de l'énergie en mécanique. En effet, un tel système est stable si son énergie mécanique totale décroit à tout instant. De ce point de départ, Lyapunov propose de construire une fonction scalaire de type énergie (nommée fonction de Lyapunov) se rapportant à l'état du sytème. Si l'énergie associée à cette fonction décroit à tout instant alors le système sera stable. Le grand avantage de cette méthode est son champ d'application général puisqu'elle permet de déterminer la stabilité d'un système quelles que soient ses caractéristiques. La principale difficulté est de déterminer la fonction de Lyapunov qui permettra de vérifier la

stabilité du système.

Une des approches classiques pour effectuer la preuve de stabilité est d'utiliser un lemme dérivé du lemme de Barbalat [SL91, Lemme 4.3]. Soit V une fonction scalaire. Le système est stable si et seulement si :

- V a une borne inférieure. Ainsi si l'on choisit V comme une fonction de Lyapunov, c'est-à-dire de type énergie, alors $V \geq 0$ et elle est donc forcément bornée.
- $\dot{V} < 0$
- $\dot{V}$ est uniformément continue en fonction du temps. Cette dernière condition se traduit généralement par la vérification que $\ddot{V}$ est bornée.

En plus de permettre l'étude de la stabilité d'un système, l'utilisation de la méthode de Lyapunov permet également de synthétiser des contrôleurs. En effet, partant du système à contrôler et du comportement désiré, il est possible d'utiliser une fonction de Lyapunov pour asservir le système sur le comportement désiré. Il s'agit de choisir une fonction de Lyapunov portant sur l'état du système puis d'appliquer une loi de commande qui permette à cette fonction de respecter les critères de stabilité évoqués ci-dessus. Les lois de commande conçues suivant cette approche sont, par contre, monolithiques.

Toutefois, ces approches se basent sur une exécution en temps continu. Or, à l'instar de ce que nous avons présenté précédemment pour les systèmes linéaires, la notion de stabilité en temps discret doit être complétée par la détermination d'un ensemble de périodes d'exécution permettant de garantir cette stabilité.

Dans ce contexte, les approches utilisables sont très peu nombreuses. En effet, la grande majorité des travaux proposés dans le cadre de l'étude de stabilité des systèmes non linéaires (tels que [AI93] ou [BB01]) en temps discret se basent sur un système discret commandé par un contrôleur discret. Cela leur permet d'appliquer une extension du théorème de Lyapunov pour le temps discret. Dès lors, la période de fonctionnement du contrôleur n'a plus d'impact. Mais cette approche ne correspond pas au cas applicatif rencontré en robotique où le robot est un système continu et le contrôleur exécuté en temps discret.

Certaines approches telles que [BFNP13] (pour la commande de robots industriels), [SMKR11] (pour la commande de flotilles de robots avec des communications incertaines) ou [ZBP01] (dans le cas où le système et son contrôleur sont reliés par un réseau générant des délais) étudient la stabilité sans passer par la transformée en z. Elles sont malheureusement limitées à des modèles linéaires mais pourraient servir de base au développement d'approches pour les

systèmes non linéaires.

Une approche permettant d'évaluer les périodes d'exécution garantissant la stabilité, similaire à la méthode indirecte de Lyapunov, peut consister à linéariser le système autour d'un point de fonctionnement pour ensuite appliquer les outils d'analyse de ce domaine. Bien sûr, cette approche est limitée par son domaine de validité.

Il ne faut également pas négliger l'intérêt de la simulation pour déterminer ces périodes. En effet, même si elle n'a pas force de preuve, elle permet une estimation des ensembles de périodes d'exécution pour lesquelles notre système est stable. Comme toutes les méthodes présentées précédemment, cette approche est naturellement dépendante de la précision de modélisation du système puisque la stabilité à une période donnée dépend des paramètres du système.

2.2 Description modulaire d'une loi de commande

Si la modularité n'est qu'une préoccupation relativement récente pour les automaticiens, certaines approches ont déjà tenté d'intégrer cette problématique. Nous présenterons dans cette section les approches qui, tout en tentant d'amener une plus grande modularité dans la conception des lois de commande, ont cherché à ce que les solutions proposées restent compatibles avec les outils traditionnels de l'automatique (notamment, comme évoqué précédemment, pour l'étude de la stabilité).

2.2.1 Contrôle avec commutations

Les approches basées sur le contrôle avec commutations (Switching control) et les systèmes hybrides telles que [TRC+10] permettent d'introduire de la modularité dans la conception du contrôle. Nous présenterons dans un premier temps des considérations générales sur de tels systèmes, notamment liées à la preuve de leur stabilité, puis un exemple de contrôle appliqué à la robotique mobile illustrera ces principes.

Généralités sur les systèmes hybrides et le contrôle avec commutations

Les explications présentées ici sont basées sur [Lib03].

Un système hybride est un système comprenant une interaction entre une dynamique continue et une dynamique discrète.

Considérons un système continu dont la dynamique est décrite par l'équation $\dot{x} = Ax+Bu$ avec $x \in \mathbb{R}^n$ l'état du système et $u \in \mathbb{R}^m$ ses entrées de commande. Considérons également un automate à état-fini d'état q, les valeurs possibles pour q étant un ensemble fini $\mathbb{Q}$. Les

transitions entre deux états de $\mathbb{Q}$ sont déclenchées par une entrée v, dont les valeurs sont dans l'ensemble $\mathbb{V}$, selon la fonction de transition d'état $t : \mathbb{Q} \times \mathbb{V} \to \mathbb{Q}$.

Ce système sera défini comme hybride si $u = f(q)$ et $v = g(x)$, f et g étant des fonctions quelconques. La Figure 2.6 résume graphiquement la définition ci-dessus.

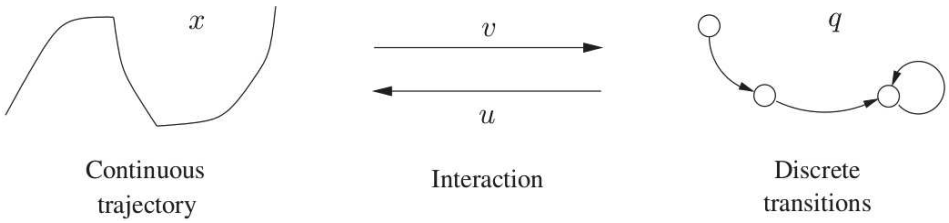

FIGURE 2.6 – Un système hybride (figure tirée de [Lib03, Figure 1])

En théorie du contrôle, l'étude se focalise principalement sur les propriétés du système continu, lequel est considéré comme étant associé à des évènements de commutation discrets. De tels systèmes, où le comportement du système discret est négligé, sont appelés systèmes commutés (switched systems). L'une des principales préoccupations lors de l'étude de tels systèmes à des fins de contrôle est de déterminer leur stabilité et les conditions que celle-ci impose sur les évènements de commutation.

Se basant sur la théorie précédente, le contrôle avec commutations consiste à décrire le contrôleur non pas comme un unique contrôleur mais comme un ensemble de n contrôleurs couplés à une logique décisionnelle chargée de déterminer le contrôleur à utiliser. Ceci est représenté dans la Figure 2.7, où y représente les capteurs placés sur le système. Dans la suite, le signal de commutation sera noté σ afin de rester cohérent avec les notations de la littérature.

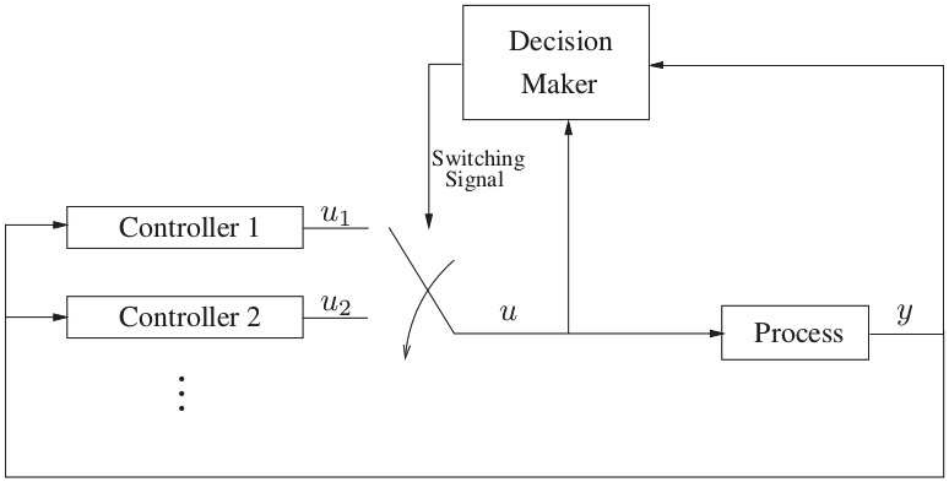

FIGURE 2.7 – Un contrôle avec commutations (figure tirée de [Lib03, Figure 20])

Evidemment l'étude de la stabilité de tels systèmes est un point crucial. En effet, même si chaque contrôleur est lui-même stable, les commutations peuvent générer une instabilité du système. Deux approches majeures se dégagent pour étudier la stabilité de tels systèmes et établir les conditions s'appliquant sur le signal de commutation pour que les commutations ne provoquent pas d'instabilité. Il faut néanmoins noter que ces approches se basent sur l'étude de la stabilité en temps continu (notamment les fonctions de Lyapunov précédemment présentées). L'impact d'une exécution en temps discret et de la période d'exécution n'est pas mis en lumière dans ces approches.

Les fonctions de Lyapunov communes :

Si les différents contrôleurs partagent une fonction de Lyapunov commune alors le contrôleur est Globalement Uniformément Asymptotiquement Stable (GUAS).

Ici la preuve de stabilité ne diffère pas de celle des contrôleurs ne présentant pas de commutation. En effet quel que soit le contrôleur utilisé, l'état du sytème continue à évoluer selon la fonction de Lyapunov tendant donc à se stabiliser.

De plus, dans ce cas, la stabilité est indépendante du signal de commutation choisi.

Dwell Time :

Si une fonction commune ne peut être trouvée, alors il faut s'assurer que la période du signal de commutation soit suffisamment élevée pour permettre aux effets transitoires générés lors de la commutation de se dissiper.

Ainsi, si l'on introduit un temps $\tau_d > 0$ tel que, quels que soient deux instants de commutation consécutifs t_i et t_{i+1}, l'inégalité $t_{i+1} - t_i \geq \tau_d$ soit satisfaite, le contrôleur est stable si τ_d est suffisament grand. τ_d est appelé *dwell time*. L'objectif de cette approche est donc de déterminer la borne inférieure de τ_d afin que la stabilité du système soit garantie. En outre, elle nous permet d'exprimer des contraintes temporelles sur le signal de commutation.

Néanmoins, le concept de *Dwell Time* peut s'avérer trop restrictif notamment dans le cadre des applications robotiques. On pourra alors considérer son extension, baptisée *Average Dwell Time* [HM99]. Dans ce cadre, on définit un nombre maximal de commutations autorisées sur une fenêtre glissante de temps. Ainsi, des commutations plus rapides seront possibles quand l'application le nécessitera (pour par exemple prendre en compte une panne sur le robot, ou la rencontre d'un obstacle) puis il sera possible de compenser en ralentissant la période des commutations par la suite afin de conserver l'écart moyen entre les commutations.

Une application de contrôle avec commutations

Considérons l'exemple de contrôle avec commutations présenté dans [TRC+10]. Celui-ci illustre comment une approche basée sur les commutations permet d'introduire de la modularité dans la conception du contrôle.

Cet exemple présente un algorithme de suivi d'un contour quelconque basé sur le principe de contrôle avec commutations. Si le problème d'un suivi de mur de forme régulière (i.e. ne présentant pas de variations soudaines dans sa forme, voir Figure 2.8.a) est relativement simple à résoudre, la présence d'une variation abrupte soit due au contour lui-même (ce qui risque d'entraîner la "perte de contact" avec le contour, voir Figure 2.8.b) ou due à un coin intérieur au contour (risque de collision avec la partie du mur faisant face au robot, voir Figure 2.8.c) est difficile à traiter avec un contrôleur unique.

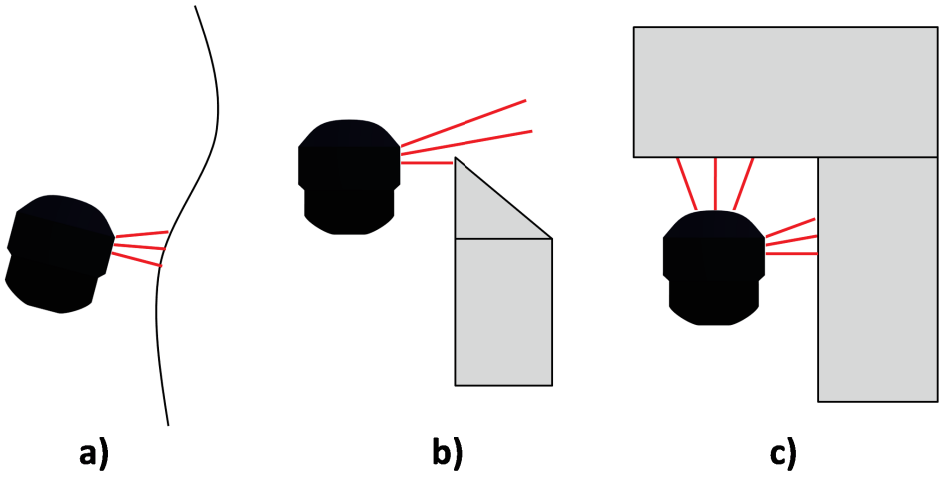

FIGURE 2.8 – Trois situations que peut rencontrer le robot lors d'un suivi de contour

Ainsi cette approche permet de décomposer le problème global en trois sous-problèmes et de développer de manière indépendante une solution à chaque sous-problème apportant de la modularité dans la conception du contrôleur. La logique de commutation se base quant à elle sur deux signaux générés à partir des mesures sonar. La Figure 2.9 décrit le contrôleur ainsi obtenu.

Wall-Following est le contrôleur de suivi de mur. *Circular Path* représente le contrôleur gérant la situation de "perte de contact" avec le contour. Il consiste à faire décrire au robot un cercle de rayon $R = \tilde{d}_{lost}$ où $\tilde{d}_{lost}$ est la distance au mur avant la perte de contact. Enfin, *Orientation* correspond au contrôleur pour traiter les collisions. Il fait simplement tourner le robot sur lui-même jusqu'à ce qu'il fasse face à une zone sans obstacle.

La preuve de stabilité du contrôleur est réalisée en utilisant la méthode des fonctions de Lyapunov communes.

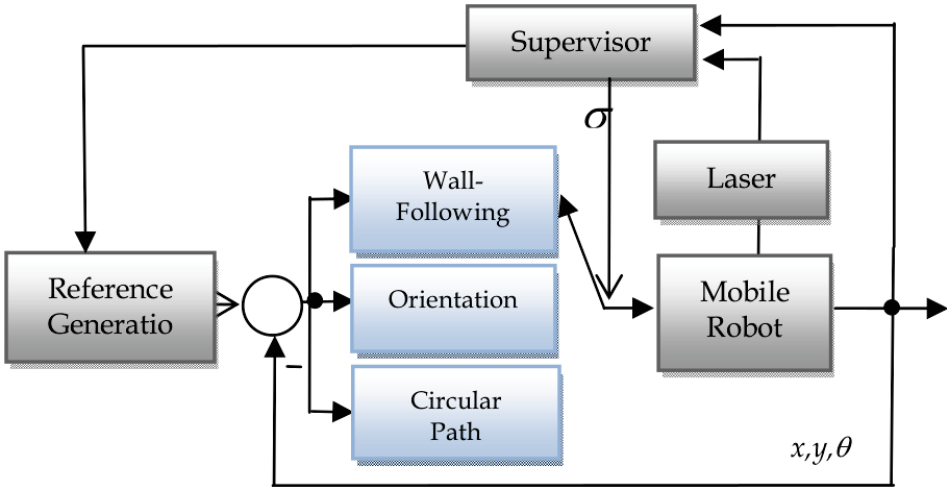

FIGURE 2.9 – Contrôleur avec commutations pour le suivi de contour (figure tirée de [TRC+10, Figure 11])

Limitations de cette approche

Néanmoins cette approche présente un certain nombre de limitations. En effet, afin de pouvoir étudier la stabilité du système, il est nécessaire que chaque contrôleur possède les mêmes états. Cela limite naturellement les contrôles descriptibles via cette approche.

De plus dans le cadre d'un système comme celui décrit dans la Figure 2.7, la transition entre les contrôleurs lors de la commutation n'est pas considérée. Ainsi l'initialisation du contrôleur démarré (celui qui va devenir actif) en fonction du contrôleur qui est interrompu (celui qui était précedemment actif) n'est pas présentée. En effet, dans cette approche, les contrôleurs sont considérés comme évoluant "en parallèle" et donc la commutation n'est vue que comme une modification de la liaison entre les contrôleurs et le processus contrôlé comme cela est illustré dans la Figure 2.7. Mais bien sûr il ne s'agit pas d'une solution viable dans le cadre d'une implémentation et il faut alors se préoccuper de l'initialisation de l'état du contrôleur à activer en fonction de l'état du contrôleur interrompu comme illustré par exemple dans [Car04].

Dans [TRC+10], cette problématique n'est pas non plus explicitement traitée. Néanmoins, l'action d'initialiser le rayon du cercle à effectuer, dans le cadre du contrôle choisi pour gérer la perte de contact avec le contour, à partir de la dernière distance par rapport au mur suivi, dénote une prise en compte de cette problématique ainsi qu'une ébauche de solution.

Enfin dans [TRC+10], si la stabilité "théorique" du système est prouvée, l'implémentation

du contrôle sur le robot et les problématiques qui lui sont associées ne sont pas considérées.

2.2.2 Modularité entre contrôleur et loi de mise à jour des paramètres

Contrôler un robot nécessite souvent d'utiliser certaines de ses caractéristiques (sa masse par exemple) ainsi que des propriétés de son interaction avec l'environnement (coefficients de frictions) comme paramètres du contrôle. Néanmoins l'estimation de ces paramètres, notamment dans le cadre de l'évolution du robot dans un environnement inconnu ou changeant comme le milieu sous-marin, étant complexe, il est nécessaire d'ajuster leur valeur en ligne.

Ainsi un contrôle est souvent associé avec une loi d'adaptation des paramètres. Néanmoins comme souligné dans [DdQDF04], les deux sont souvent conçus ensemble en utilisant une seule et même fonction de Lyapunov. La marge de manœuvre du concepteur est donc limitée et un changement de loi d'adaptation nécessite de déterminer une nouvelle fonction de Lyapunov afin d'étudier la stabilité de l'ensemble.

Pour répondre à ces limitations, [DdQDF04] étend aux robots mobiles l'approche basée estimation, proposée dans [dQDA99] pour les robots manipulateurs. Ainsi, suivant cette approche toute loi d'adaption des paramètres peut être utilisée à condition qu'elle satisfasse les conditions suivantes :

- L'estimation des paramètres reste bornée.
- L'erreur de prédiction doit être intégrable au carré.
- La matrice d'inertie estimée doit être définie positive (la loi d'adaptation des paramètres doit être un algorithme de type projection).

De ce fait une plus grande marge de manœuvre est laissée pour la conception de la loi d'adaptation. Pour cela le contrôle doit être conçu afin de respecter les propriétés relatives à la stabilité *Input-to-State* (ISS). Ce faisant, il est possible de déterminer que les erreurs de contrôle convergent globalement et asymptotiquement vers une valeur constante arbitraire pouvant être rendue aussi faible que souhaitée.

Néanmoins cette approche reste difficilement généralisable à d'autres problématiques de contrôle. L'implémentation des lois de commande n'est également pas considérée explicitement. Ainsi dans [dQDA99], bien que les périodes de contrôle choisies soient indiquées, aucune explication précise n'est donnée quant à leur choix sinon que ce choix se base sur la complexité des calculs.

2.2.3 Les approches basées sur le Motion Description Language

La réutilisabilité d'un contrôleur d'une cible d'implémentation à une autre est un élément clé pour gagner en efficacité dans le développement des lois de commande. En plus d'une conception modulaire, cela nécessite également de réaliser une description de la loi de commande à un niveau d'abstraction suffisant pour être indépendante des caractéristiques du système cible. Cela nécessite généralement d'utiliser un langage spécifique qui permet cette abstraction. Néanmoins, il faut faire attention à ce que ce langage puisse être ensuite transcriptible sous forme de logiciel de contrôle tout en conservant des performances d'exécution similaires à celles obtenues par une conception spécifique à la cible d'implémentation.

Pour résoudre cette problématique dans le cadre du contrôle d'un bras manipulateur, Roger Brockett propose une approche baptisée *Motion Description Language* (MDL) [Bro88]. Il propose d'écrire l'entrée de contrôle $v(t)$ sous la forme :

$$v(t) = u(t) + k(y(t))$$

avec $u(t)$ le terme de contrôle en boucle ouverte et $k(y(t))$ le terme d'ajustement en boucle fermée où $y(t)$ est l'observation des états du système.
Ce faisant, il introduit la notion de *segment modal* comme étant le triplet (u, k, T) où T définit la durée d'application du *segment modal*. Ainsi le contrôle d'un robot peut être décrit comme une succession modulaire de *segments modaux* : $(u_1, k_1, T_1)...(u_n, k_n, T_n)$ [Bro88].

Il n'est de plus pas rare qu'un contrôle, afin d'être plus simple à décrire, ne soit pas directement exprimé dans l'espace des actionneurs du robot. Il est nécessaire d'effectuer alors un changement de repère afin de revenir dans le repère actionneur (par exemple dans le cas des robots manipulateurs, le mouvement exprimé dans le repère effecteur doit être transformé en mouvement dans le repère des moteurs en utilisant les relations cinématiques). Pour prendre cette problématique en considération, un opérateur supplémentaire $\Phi(y(t))$ est introduit afin de permettre les changements de repère.

Néanmoins, l'une des principales limitations de cette approche est l'absence de prise en compte de la possibilité que des évènements extérieurs viennent interrompre l'exécution du *segment modal* (par exemple si le robot rencontre un obstacle). Pour résoudre ce problème, [MKH98] propose une extension des travaux sur les MDLs baptisée *Extended Motion Description Language* (*MDLe*) et applique celle-ci au contrôle des robots mobiles non-holonomes.

Ces travaux introduisent la notion d'*atome* qui étend le concept de *segment modal*. Un atome, σ, est décrit comme un triplet : $\sigma = (U, \xi^a, T^a)$.

U est l'entrée de contrôle, il s'agit d'une matrice $U = (u_1, ..., u_m)'$ avec m le nombre

d'entrées de commande du sytème avec chacun de ses éléments décrit comme l'application mathématique :

$$\begin{aligned} u_i : \mathbb{R}^+ \times \mathbb{R}^p &\rightarrow \mathbb{R} \\ (t, y(t)) &\mapsto u_i(t, y(t)) \end{aligned}$$

avec p la dimension de $y(t)$.

ξ^a est un signal d'interruption booléen basé sur un signal $s(t)$ de dimension k obtenu à partir des données capteurs. Ainsi ξ^a peut être décrit comme :

$$\begin{aligned} \xi^a : \mathbb{R}^k &\rightarrow \{0, 1\} \\ s(t) &\mapsto \xi^a(s(t)) \end{aligned}$$

T^a est la durée maximale d'exécution telle qu'elle avait été définie pour un *segment modal*. L'*atome* s'exécutera donc jusqu'à ce que soit ξ^a passe de 1 à 0 soit jusqu'à ce que $t - t_{start} \geq T^a$ avec t_{start} l'instant où l'exécution de l'*atome* a démarré.

Les *atomes* peuvent ensuite être composés en différentes chaînes. Les *comportements* (*behaviors*) ne regroupent que des *atomes*, un *comportement* b constitué de deux atomes σ_1 et σ_2 est décrit par le triplet : $b = ((\sigma_1, \sigma_2), \xi^b, T^b)$. *Comportements* et *atomes* peuvent être regroupés en *plans partiels* (*partial plans*) eux-mêmes associés en plans [HVKA+00].

De plus le langage autorise deux opérateurs supplémentaires pour la construction de chaînes :

- Un opérateur de mise à l'échelle : $(\alpha, \beta)(U, \xi^a, T^a) = (\alpha U, \xi^a, \beta * T^a)$ avec $\alpha \in \mathbb{R}^m$.
- Un opérateur permettant de réaliser des boucles noté $(\sigma_1, \sigma_2)^n$ qui indique que la chaîne (σ_1, σ_2) est répétée à n reprises, n pouvant être infini.

[MKH98] propose également une architecture permettant de structurer l'utilisation des différents éléments. Il s'agit d'une architecture hybride constituée de deux niveaux (Figure 2.10).

Le *planificateur* est chargé de produire les *plans partiels* permettant au robot de réaliser les mouvements souhaités afin d'atteindre les objectifs de mission en fonction de l'environnement existant. Les plans sont produits par composition des *comportements* existants.

Le *système moteur* (*Motor System*) est chargé de faire le lien entre la représentation symbolique manipulée par le *planificateur* et les *machines à état cinématique* (*Kinematic State Machines*) qui implémentent les *atomes*. Une d'elles est représentée à la Figure 2.11. Dans ce cas, elle suppose que $y(t) = x(t)$.

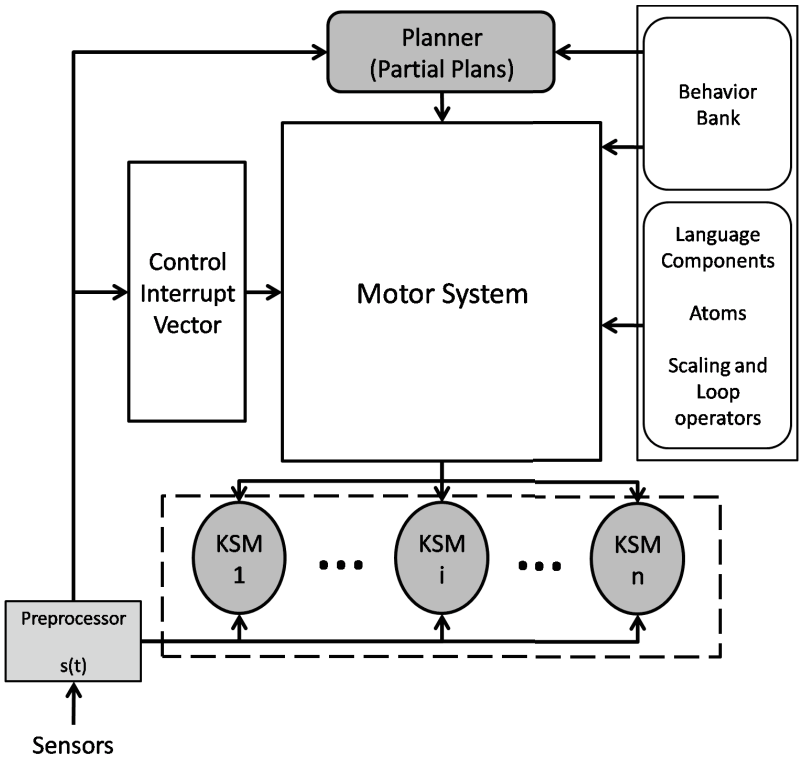

FIGURE 2.10 – Architecture Hybride associée au *MDLe* (figure tirée de [MKH98, Figure 6.4])

Le préprocesseur est chargé de produire le signal $s(t)$ en fonction des entrées capteurs. L'activation du contrôle $U(t, x)$ n'est possible que si les conditions (i.e. signal d'interruption et durée d'exécution) d'activation de l'*atome* (ξ^a et T^a), du *comportement* auquel il appartient (ξ^b et T^b) et du *plan partiel* dont il fait partie (ξ^p et T^p) sont vérifiées.

Enfin dans l'architecture, le *vecteur d'interruption de contrôle* (*control interrupt vector*) permet à un utilisateur de forcer l'inhibition d'un *atome*, *comportement* ou *plan partiel*.

Les *MDLes* proposent donc une approche afin d'articuler d'un côté la conception des lois de commande et de l'autre leur implémentation sous forme de logiciel de contrôle. Pour cela, elle propose une description de la suite des actions à réaliser à un niveau d'abstraction suffisamment élevé pour assurer un cloisonnement étanche entre les problématiques générales de contrôle et les spécifités du système cible.

Cette approche présente toutefois certaines limitations. Ainsi si un contrôle peut être décomposé en une succession d'appels à des lois de commande, ces dernières demeurent conçues de manière monolithique. Les modèles que ces lois utilisent ne sont ainsi pas explicités. De plus une autre limitation importante est la séquentialité du langage. En effet, lorsque les conditions d'arrêt d'un *atome* sont vérifiées c'est l'*atome* suivant dans la chaîne qui s'exécute. Des

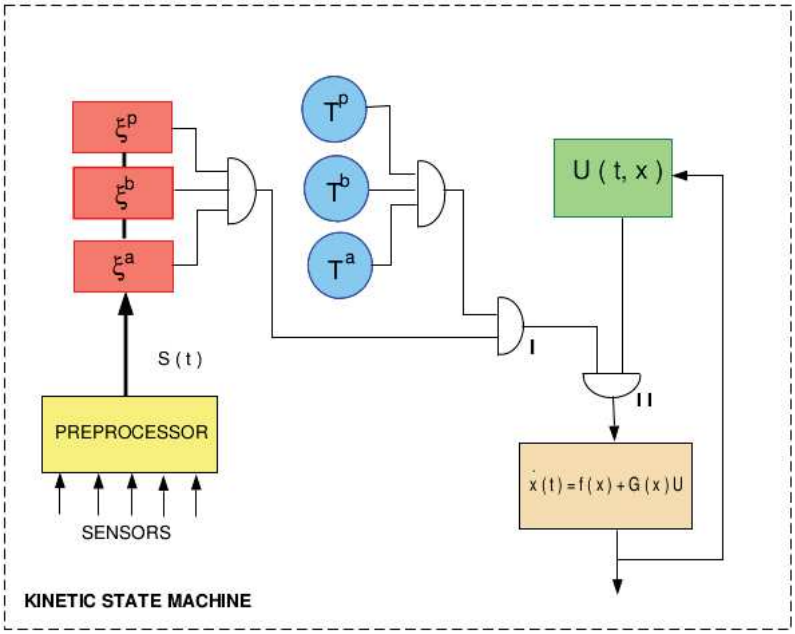

FIGURE 2.11 – Une machine à état cinématique

séquences telles que celles de la Figure 2.12 ne sont donc pas descriptibles avec les *MDLes*. Cette difficulté peut néanmoins être contournée en utilisant les conditions d'exécution des *comportements* et *plans* mais cela nécessite donc de porter une attention toute particulière à la structuration de ces derniers. Qui plus est, cette approche ne permet pas d'exprimer une exécution parallélisée de plusieurs *atomes*.

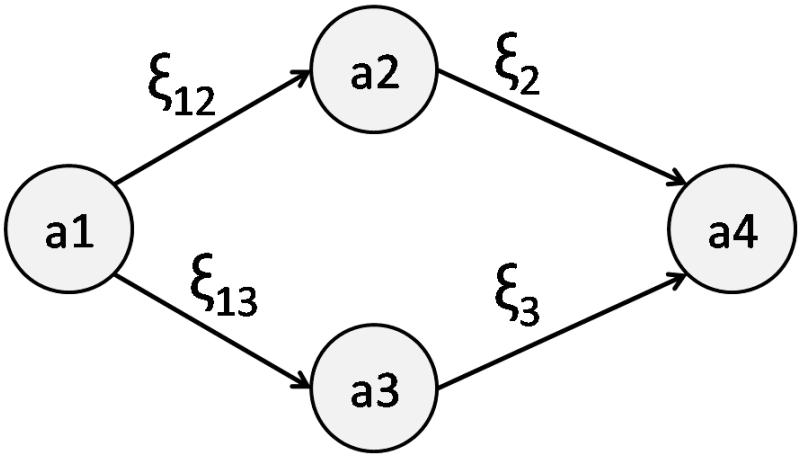

FIGURE 2.12 – Une structure difficilement réalisable avec un *MDLe*

Enfin dans [HVKA+00], un *framework*, le *MDLe Engine*, est proposé pour permettre d'interpréter le langage et de produire le code C/C++ (avec des points d'interfaçage avec les *drivers* spécifiques à la cible d'implémentation) exécutable associé aux différents *atomes*. Ce

framework comprend également un ordonnanceur chargé de gérer l'exécution des différents *atomes*, *comportements* et *plans* afin de respecter leurs durées d'exécution. Il propose également de diviser les *atomes* en deux niveaux de priorité. Néanmoins, les durées d'exécution imposées aux différents composants ne sont pas explicitement reliées aux problématiques d'implémentation de ceux-ci et aux contraintes qui en découlent.

Une autre limitation de ces approches est l'absence de prise en compte des problématiques de stabilité engendrées par la commutation entre les différentes stratégies de commande.

2.3 Modélisation de systèmes et modularité

Les phénomènes environnementaux et les systèmes d'origine humaine (processus industriel, véhicule, robot) nécessitent d'être étudiés et analysés afin d'en comprendre le fonctionnement précis pour à la fois en tirer le plein potentiel mais aussi pour pouvoir anticiper les problèmes qui peuvent émerger du fonctionnement et, si possible, les résoudre. Pour cela les ingénieurs doivent développer des modèles mathématiques à partir de leurs connaissances sur les systèmes. Ces modèles doivent ensuite être transformés en artefacts informatiques utilisés notamment pour simuler ces systèmes, étape indispensable pour leur compréhension et également pour valider les modèles.

Les langages de modélisation permettent de faciliter la traduction des modèles mathématiques en entités informatiques permettant leur exploitation. Or de plus en plus de systèmes sont hétérogènes. Ils peuvent comprendre des composants provenant de domaines très divers (par exemple les systèmes cyber-physiques qui comprennent entre autres des composants électroniques, mécaniques, des processus chimiques ou des réseaux). Les modèles de ces systèmes sont en outre développés dans des disciplines mathématiques différentes qui ont chacune leur propre paradigme de représentation. Enfin, l'évolution des connaissances théoriques nécessite souvent que les modèles soient enrichis ou corrigés. De fait la modularité est devenue une préoccupation importante lors de la modélisation d'un système surtout lorsque celui-ci est hétérogène.

Si la conception de contrôleurs n'est pas la préoccupation première de ces approches, les mécanismes qu'elles proposent s'avèrent pertinents pour notre problématique.

2.3.1 Simulink

Ainsi un premier effort de standardisation, CSSL, apparut en 1967. Il propose une méthodologie qui consistait à décrire des systèmes sous forme de blocs d'entrées/sorties. L'outil le plus répandu basé sur cette approche est Simulink [Kar06].

Simulink propose de décrire un système, un contrôleur ou une combinaison des deux sous forme d'un diagramme constitué de blocs d'entrées/sorties reliés par des liens. Les données échangées entre les blocs peuvent être des nombres, des vecteurs ou des matrices. Simulink permet également de s'assurer de la cohérence des dimensions au niveau des connexions entre blocs. Qui plus est, il est possible de simuler des systèmes en temps continu ou en temps discret (notamment en utilisant des blocs pour représenter les bloqueurs d'ordre 0). La grande variété des blocs disponibles permet de décrire des systèmes dans une large gamme de domaines applicatifs. Enfin, des blocs permettent de décrire des sous-systèmes (des diagrammes encapsulés dans un bloc) afin de mieux structurer la description d'un système.

Néanmoins, l'approche proposée par Simulink présente un certain nombre de limitations. La principale est le manque d'expressivité de l'interface des blocs. En effet, Simulink ne permet d'exprimer que des données numériques. Il n'est ainsi pas possible d'attacher un type ou une unité aux données. Cela rend les blocs très génériques et la lecture des diagrammes plus complexe. Une autre limitation, exprimée dans [EM97], est rattachée à l'utilisation de blocs d'entrées/sorties. Ce mécanisme n'est pas réellement adapté à la représentation de certains éléments (notamment des composants matériels) dont la modélisation est en conséquence difficile.

2.3.2 Modelica

Pour répondre à la problématique de description modulaire de systèmes hétérogènes, le développement d'une nouvelle approche commença en 1996. Elle aboutit au langage Modelica [EM97]. Ce langage repose sur le paradigme objet afin de bénéficier de ses apports notamment en termes de modularité et d'évolutivité. L'encapsulation offerte par le paradigme objet permet de faire cohabiter ensemble des paradigmes de représentation divers (équations différentielles, équations aux différences, automates à état fini ou encore réseaux de Petri) tout en autorisant leur manipulation de manière homogène [EMO98].

Modelica est un langage fortement orienté vers la réutilisation des composants et permet à la fois une composition hiérarchique de divers composants et une forte utilisation des mécanismes d'héritage. Il existe divers types de classes. La classe *model* sert à définir les composants ainsi que leurs assemblages. L'Exemple de Code 2.1 présente le système de contrôle moteur présenté dans la Figure 2.13 décrit dans le langage Modelica [EMO98].

Exemple de Code 2.1 – Représentation Modelica d'un système de contrôle moteur

```
1 model MotorDrive
2   Motor motor;
```

```
3    PI controller;
4    Gearbox gearbox(n=100);
5    Shaft J1(J=10);
6    Tachometer w1;
7  equation
8    connect (controller.out, motor.in);
9    connect (motor.flange, gearbox.a);
10   connect (gearbox.b, J1.a);
11   connect (J1.b, w1.a);
12   connect (w1.w, controller.in);
13 end MotorDrive;
```

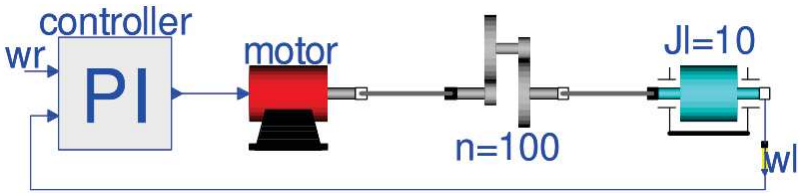

FIGURE 2.13 – Un système de contrôle moteur (figure tirée de [EMO98, Figure 1])

La première partie du modèle, lignes 2 à 6, sert à indiquer les composants du système. La seconde partie décrit le comportement du système. Ici, vu qu'il s'agit de la description d'un système, il ne contient que la description des connexions entre ses composants. Il est également à noter, comme observé lignes 4 et 5, que lors de la déclaration d'un composant, il est possible de valuer certains de ses paramètres.

Les composants eux-mêmes sont définis via un *model* comme l'illustre l'Exemple de Code 2.2 tiré de [EMO98].

Exemple de Code 2.2 – Représentation Modelica d'un arbre de transmission

```
1 model Shaft
2    extends TwoFLange;
3    parameter Inertia J=1;
4    AngularVelocity w;
5  equation
6    a.r = b.r;
7    der(a.r) = w;
8    J*der(w) = a.t + b.t;
9  end Shaft;
```

Dans la première partie du *model*, les paramètres sont définis (ici l'inertie du moteur) ainsi que les grandeurs internes au composant. Son comportement interne est décrit après le mot clé *equation* (lignes 6 à 8). Enfin le mot clé *extends* sert à indiquer que le composant dérive d'un autre composant. L'héritage est au centre du concept de réutilisation dans Modelica. En effet, il permet de définir les propriétés communes à différents composants similaires pour que le concepteur du composant puisse se concentrer sur le comportement spécifique de celui-ci. Ainsi, dans ce cas, l'arbre hérite du composant *TwoFlange*. Effectivement, un arbre de transmission a deux brides. Le composant de base est défini à l'Exemple de Code 2.3.

Exemple de Code 2.3 – Représentation Modelica d'un composant fait de deux brides

```
1 partial model TwoFlange
2   Flange a, b;
3 end TwoFlange;
```

Un composant qui sert à définir un modèle "abstrait" (équivalent aux classes abstraites proposées par exemple dans le langage C++) est défini par le mot clé *partial model*. Dans ce cas là, aucune équation interne n'est définie mais il est évidemment possible de le faire. Pour définir les classes servant à connecter deux composants ensemble, comme une *Flange* définie précédemment, cela est réalisé avec le mot clé *connector* tel que montré dans l'Exemple de Code 2.4.

Exemple de Code 2.4 – Représentation Modelica d'une bride

```
1 connector Flange
2   Angle r;
3   flow Torque t;
4 end Flange;
```

Cette classe définit les données internes au connecteur. Le mot clé *flow* sert à indiquer qu'au niveau du connecteur la somme des couples induits par différents composants doit être nulle. Les calculs et les vérifications découlant de cette hypothèse sont directement effectués par Modelica.

La gestion des types est également assurée via des classes. Les classes de *type* doivent hériter de classes prédéfinies représentant les types informatiques de base (entiers, réels par exemple) et redéfinissent certaines de leurs propriétés comme illustré par l'Exemple de Code 2.5.

Exemple de Code 2.5 – Représentation Modelica du type Angle

```
type Angle = Real(quantity = "Angle", unit = "rad", displayUnit = "deg");
```

Le paramètre *quantity* sert à spécifier la grandeur physique représentée par le type, *unit* l'unité utilisée dans les échanges d'information et *displayUnit* l'unité à afficher si différente de celle utilisée dans les connexions. D'autres paramètres que ceux-ci sont également utilisables [Mod00]. Des grandeurs physiques identiques mais exprimées dans des unités différentes ne peuvent pas être connectées ensemble.

Il est également possible de définir des classes *block* dans lesquelles les différents connecteurs sont orientés entre entrées et sorties. La partie équation d'une classe peut également être remplacée par une partie *algorithme* qui permet de décrire des comportements plus complexes. Les équations supportent également la description de comportements en temps discret ou d'opérations logiques à l'aide d'opérateurs dédiés. Des comportements hybrides peuvent être décrits à l'aide de l'opérateur *when*. Plus de détails sur les très nombreux opérateurs et classes offerts par la langage Modelica sont disponibles dans [Mod00].

Enfin, même si la conception de contrôleurs n'était pas un objectif originel du langage Modelica, plusieurs librairies ont été développées afin de permettre leur description et leur analyse telles que [BL12] ou [BOT09]. Cette dernière propose des représentations de systèmes très variées en temps continu ou discret. Elle propose également plusieurs contrôleurs de structure souvent simple (Proportionnel-Intégral-Dérivé, PID, par exemple). Elle comprend également des outils permettant l'analyse des systèmes ou l'aide à la conception des contrôleurs tels que le tracé de la réponse à différent signaux (réponse impulsionnelle ou à l'échelon notamment), la détermination des pôles d'un système ou encore des outils d'aide à la conception de Filtres de Kalman. Néanmoins, elle est limitée aux systèmes linéaires.

Modelica n'étant pas conçu pour la description de contrôleur, il présente certaines limitations dans notre contexte applicatif. Principalement, Modelica ne peut pas faire porter sur les objets de contraintes, notamment temporelles, relatives à leur implémentation. En outre, certains composants logiciels tels que des drivers ne sont pas simples à exprimer dans Modelica où les différents composants doivent avoir une équation ou un algorithme constitutif.

2.4 Conception modulaire de logiciel de contrôle en robotique

La problématique de modularité est depuis longtemps une préoccupation clé des ingénieurs logiciels dans leurs travaux. Ceux-ci ont menés au développement de nombreuses architectures de contrôle ou de *Middlewares* robotiques. Ils permettent dès lors de grandement accélérer le développement d'applications robotiques en facilitant la réutilisation de composants logiciels et en proposant une couche d'abstraction logicielle permettant de changer plus efficacement de cible d'implémentation.

Ces dernières années de très nombreuses approches ont été proposées. Nous ne présenterons que les principaux travaux réalisés dans le domaine. Pour une présentation générale des différentes approches ainsi que de leurs apports dans le contexte robotique, le lecteur pourra se référer à [BS09] et [BS10] ou encore [PACGD14].

En parallèle de ces approches, il existe des approches purement descriptives notamment les *Domain Specific Languages* (DSLs) qui ont pour objectif de représenter les concepts mis en œuvre dans les différentes applications robotiques et leurs interactions mais négligent les aspects liés à l'implémentation. Les travaux portant sur l'*autonomic computing* [SSS14] ne seront également pas détaillés. En effet, s'ils se situent à l'interface entre automatique et aspects logiciel, ils s'intéressent aux aspects temporels afin de gérer les ressources d'exécution (charge processeur, consommation énergétique) mais sont peu focalisés sur les aspects de description à un niveau d'abstraction plus élevé de l'implémentation. Nous évoquerons toutefois leur complémentarité avec nos travaux, en perspective.

2.4.1 Middlewares robotiques

Les *Middlewares* permettent de simplifier le développement d'applications logicielles en créant une couche d'abstraction entre le système d'exploitation et les applications. Dans le domaine robotique, les *Middlewares* fournissent des services permettant de simplifier la conception d'architectures logicielles. En effet, l'abstraction réalisée offre de nombreux avantages dans le développement d'architectures de contrôle ([PACGD14]) :

- portabilité/homogénéité : en travaillant à un niveau d'abstraction plus élevé, ils fournissent un modèle de programmation indépendant des spécificités du système d'exploitation utilisé. Les aspects conceptuels et les mécanismes à disposition du développeur sont donc identiques pour tous les systèmes supportés par le *Middleware*.
- efficacité : les services offerts par le *Middleware* prennent en charge de nombreux aspects bas-niveau qui sont dès lors transparents pour les utilisateurs qui ne sont généralement

pas familiers avec les aspects système, notamment temps-réel. Bénéficiant qui plus est de l'homogénéité précédemment évoquée, le développement d'applications, leur évolution ou leur portage sur une autre cible technologique s'en trouvent simplifiés et leur temps de développement réduit.

– fiabilité : les *Middlewares* (temps-réel) sont développés et testés par des spécialistes de la programmation système, temps-réel ou encore de la gestion des réseaux. Le bon fonctionnement est donc assuré pour les utilisateurs non-spécialistes. En outre, pour les *Middlewares* les plus communément utilisés, une communauté importante assure généralement leur maintenance sur le long terme et une détection rapide de tout problème pouvant apparaître.

Dans nos travaux, nous utiliserons le *Middleware* temps-réel ContrACT développé au LIRMM. Les grands principes de son fonctionnement seront présentés au chapitre 7.

Parmi les *Middlewares* robotiques les plus utilisés, nous pouvons citer ROS [QGC+09]. Développé initialement par la société Willow Garage, il est structuré autour de l'exécution de processus fonctionnant de manière indépendante, potentiellement sur des hôtes différents (au sens de nœuds différents d'un réseau), et communiquant ensemble via une topologie *peer-to-peer*. Chaque processus calculatoire est appelé un *node*. Ceux-ci communiquent entre eux en échangeant des *messages*. Ces derniers sont constitués de structures aggrégant des types de données "primitifs" offerts par ROS. Il est également possible d'échanger des tableaux de données mais aussi de définir un *message* comme un tableau de *messages* ou encore en composant plusieurs types de *messages* offrant ainsi un cadre très souple pour définir les données échangées. La communication en elle-même est réalisée suivant un mécanisme *publish/subscribe* et organisée autour du concept de *topics*. Il s'agit d'un "bus" identifié de manière unique chargé de réceptionner les *messages* des *nodes* qui publient sur ce *topic* et de transmettre les informations à ceux qui y ont souscrit. Les *messages* échangés sur un *topic* donné doivent avoir une structure clairement établie lors de sa définition. Si ce mécanisme permet une grande souplesse et un découplage entre publication et souscription, il n'est pas adapté aux communications synchrones où un accusé de réception est nécessaire. Ce type d'échange est également implémentable sous ROS en utilisant les *services* qui sont définis par trois informations, un identifiant unique, le type de *message* utilisé pour la requête et le type de *message* utilisé pour y répondre.

La structure d'un logiciel conçu avec ROS peut être représentée comme un graphe dont les nœuds sont les *nodes* et les arcs représentent les liaisons effectuées entre eux via les *topics* et *services*. Un tel graphe est, par exemple, représenté Figure 2.14. Celui-ci est affiché à l'aide de l'utilitaire *rxgraph* qui permet de le visualiser et d'afficher toutes les informations relatives aux connexions possibles avec un *node* donné.

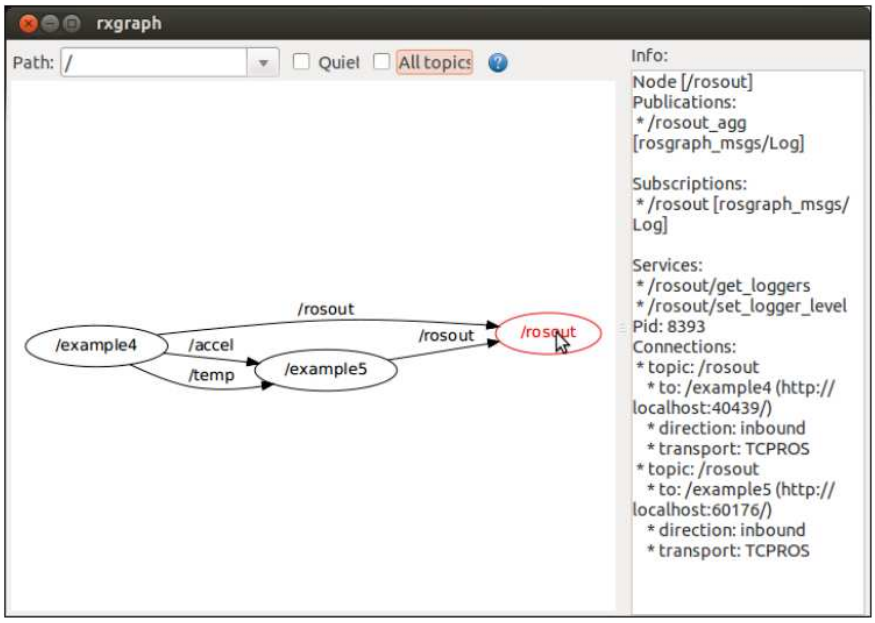

FIGURE 2.14 – Un graphe représentant la structure d'un logiciel de contrôle basé sur ROS (figure tirée de [MF13, page 98])

ROS intègre également une problématique importante, la gestion des repères dans lesquels sont représentées les différentes données. En effet, toute information qui a pour grandeur une position, une distance ou une grandeur dérivée (vitesse, accélération) est fondamentalement exprimée par rapport à un repère précis (le repère monde, le repère spécifique à un robot ou encore le repère spécifique à un actionneur). Pour résoudre ce problème, ROS propose la librairie *tf* [Foo13] chargée d'assurer le suivi des différents repères définis dans l'application et des relations de transformation (potentiellement dynamiques) entre eux. S'inspirant du principe des graphes de scène utilisés dans la visualisation 3D, elle représente les différents repères sous la forme d'un arbre (préféré à un graphe pour des raisons d'efficacité dans son parcours) dont chaque nœud correspond à un repère et les arcs représentent les transformations à effectuer pour passer de l'un à l'autre. En outre, afin de gérer les problématiques inhérentes aux architectures distribuées (délais, perte d'information), l'évolution des différentes transformations est historisée et associée à une date de validité. Ainsi quand une donnée est soumise pour un changement de repère, celle-ci est datée afin de permettre de déterminer la transformation adéquate à appliquer. Implémentée dans des *nodes* de ROS, cette librairie est facilement

associable au reste de l'architecture.

La décomposition du logiciel de contrôle en différents *nodes* permet une conception modulaire et une grande évolutivité. En effet, il est possible de voir les *nodes* comme des "modules" encapsulant un code interne et communiquant vers d'autres "modules" via une interface qui est représentée ici par les *messages* et les *topics*. La structure du système de fichiers de ROS permet également de facilement échanger des solutions logicielles entre utilisateurs. Malgré tout, même s'il facilite l'intégration de contributions diverses, l'une des principales limitations de ROS est son absence de gestion des aspects temps-réel qui sont délégués au développeur dans la conception du code interne à chaque *node*. De fait, tous les aspects temporels liés à l'implémentation d'une loi de commande sont laissés à la responsabilité du concepteur.

Un autre *Middleware* ayant connu un large succès dans la communauté robotique est Orocos (*Open Robot Control Software*). Il s'agit d'un *Middleware* temps-réel visant à favoriser le découplage entre les différents composants logiciels du système afin de favoriser la modularité et la réutilisabilité [BSK03]. Orocos est organisé autour de trois éléments :

- *RealTime Toolkit* (RTT) : un ensemble de librairies fournissant une abstraction du système d'exploitation permettant d'implémenter les différents composants du *Middleware*. Elle sert ainsi à compiler et exécuter l'application conçue par l'utilisateur comme montré Figure 2.15.
- *Orocos Component Library* (OCL) : Un ensemble de composants d'infrastructure basés sur RTT et nécessaires pour faire fonctionner l'application. Elle contient ainsi les composants permettant de paramétrer et debugger l'application, de logger les données ou encore d'assurer le bon fonctionnement des connexions entre composants.
- Une *Toolchain* permettant aux utilisateurs de développer leurs propres composants en utilisant RTT et ayant notamment la capacité de générer des squelettes de code à compléter par le code spécifique à l'utilisateur.

Un composant Orocos est appelé un *TaskContext* du nom de la classe définissant la structure de base commune à tous les composants. L'utilisateur va pouvoir définir ses propres composants en héritant de la classe *TaskContext*. Celle-ci est présentée Figure 2.16.

Le comportement interne d'un *TaskContext* est régi par une machine à états dont les transitions d'états correspondent aux fonctions qui seront exécutées. Durant la création du composant, celui-ci se trouve dans l'état *Init*. L'utilisateur peut alors choisir soit de mettre un module en état *PreOperational* rendant obligatoire sa configuration (fonction *configure()*) ou alors de démarrer directement le module dans l'état *Stopped* qui signifie que le module est prêt à être démarré. Dans le second cas, la configuration est optionnelle mais reste toujours

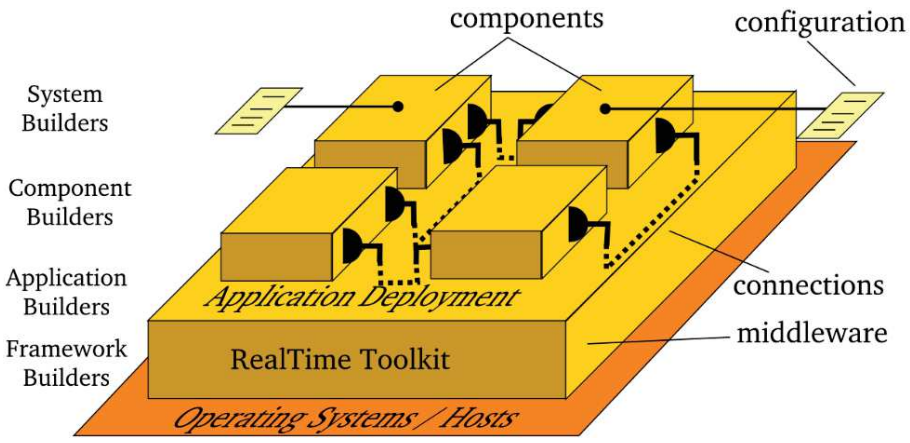

FIGURE 2.15 – La suite d'outils mis en œuvre dans le *Middleware* Orocos (figure tirée de [Soe12, Figure 1.1])

possible dès lors que le module est arrêté. Enfin, lorsque le module est démarré (via la fonction *start()*) il se retrouve dans l'état *Running* qui permet d'appeler de manière périodique ou évènementielle (suivant la configuration du module) la fonction *update()*. Enfin, il est possible de stopper l'exécution d'un module (fonction *stop()*) voire d'annuler sa configuration (fonction *cleanup()*). Chacune de ces fonctions fournit un point d'interface aux utilisateurs afin qu'ils puissent implémenter le code spécifique à chaque composant. Les *TaskContexts* sont connectés ensemble via leurs ports d'entrée/sortie. Les échanges sont réalisés suivant un flux de données. Il est à noter qu'Orocos permet l'échange d'objets et que l'exécution d'un module (appel à la fonction *update()*) peut être déclenchée sur réception d'une donnée d'un port.

Un ensemble de propriétés permet également de configurer le composant. Enfin, chaque composant peut fournir un ensemble de services appelés *Operations* qui peuvent être appelées soit par l'opérateur soit par un autre *TaskContext*. Cet appel se fait via des objets spéciaux, les *OperationCallers*. Ces *Operations* peuvent être soit des actions associées à des méthodes appartenant au composant (pour contrôler son exécution ou obtenir/modifier la valeur d'un paramètre) soit des appels à des fonctions externes.

La structuration en composants proposée par Orocos permet donc une grande modularité dans l'application tout en assurant un support de l'exécution temps-réel. De plus, Orocos n'est pas rattaché à une structuration architecturale particulière et l'échange d'objets laisse aux utilisateurs une grande liberté dans la spécification de l'interface entre composants. Cela nécessite néanmoins de mettre en place une couche d'abstraction supplémentaire qui aurait pour rôle de définir la structure de l'application et la spécification des interfaces entre *TaskContexts* afin d'assurer la cohérence de l'application.

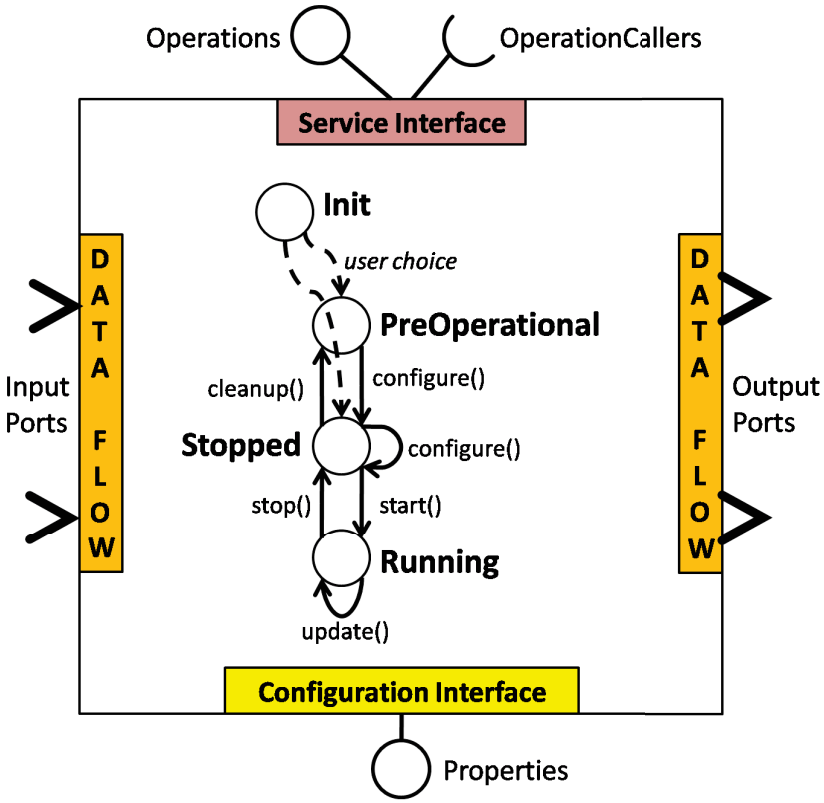

FIGURE 2.16 – Structure d'un composant Orocos

Enfin, nous pouvons également citer le *Middleware* MOOS-IvP [BSNL13]. Cette approche vise principalement à faciliter le développement d'architectures de contrôle pour les véhicules sous-marins dans le contexte d'une architecture calculatoire distribuée. L'élément central de l'approche MOOS est une base de données, la MOOSDB, qui permet aux différents processus formant une "communauté" MOOS de communiquer ensemble. Une telle "communauté" est illustrée Figure 2.17.

Les différentes applications MOOS communiquent avec la base de données via un mécanisme de publication/souscription. Ainsi, l'interface d'une application est définie comme l'ensemble des messages publiés et l'ensemble des messages auxquels l'application souscrit. Cette approche permet une grande modularité dans la conception et une grande évolutivité puisque chaque application peut être modifiée sans affecter le reste du logiciel de contrôle tant qu'elle conserve la même interface. Grâce à la base de données qui assure un découplage entre les différentes applications, celles-ci peuvent s'exécuter indépendamment et, par exemple, n'ont pas à fonctionner à la même période (l'accès partagé à la base de données partagée restant un point critique comme dans tout approche basée sur un tableau noir).

Une application MOOS notable est le système de prise de décision IvP *Helm*. En fonction des informations fournies par la base de données MOOS (par exemple des informations

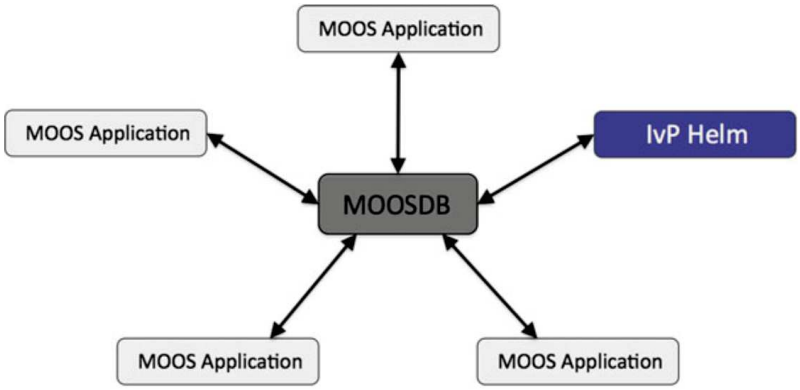

FIGURE 2.17 – Une "communauté" MOOS organisée autour d'une base de données (figure tirée de [BSNL13, Figure 2.3])

venant des capteurs ou de l'exécution d'autres applications), celle-ci va arbitrer les calculs effectués par un certain nombre de comportements (suivre un ensemble de points de passage, éviter des collisions) et transmettre différentes consignes que devront appliquer les autres applications (par exemple cap ou vitesse d'avance désirés) [BSNL13]. La Figure 2.18 illustre le fonctionnement d'IvP.

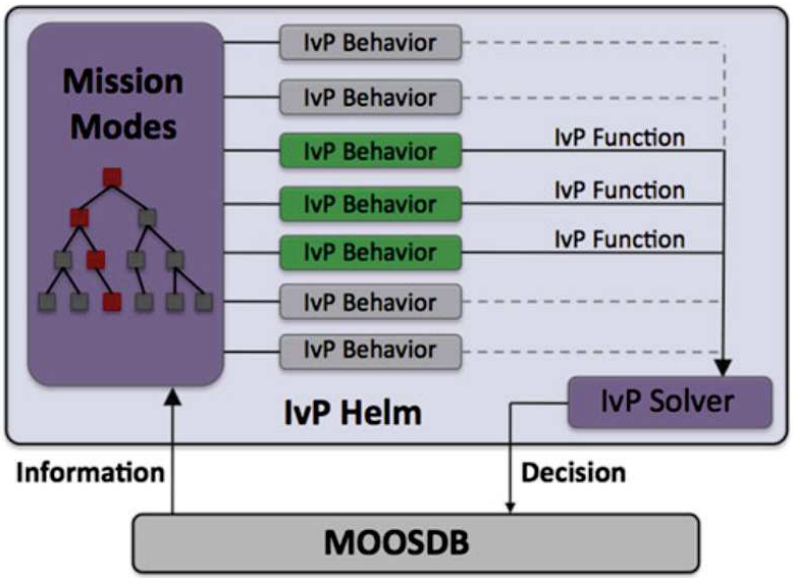

FIGURE 2.18 – Le fonctionnement du système décisionnel IvP (figure tirée de [BSNL13, Figure 2.7])

En fonction des objectifs de mission, les comportements adéquats sont choisis puis leurs sorties sont arbitrées en utilisant les principes de la programmation par intervalles afin de déterminer les consignes à appliquer [Ben04].

Ainsi, le *Middleware* MOOS-IvP permet une grande souplesse dans la conception du logiciel de contrôle d'un robot sous-marin. La décomposition en applications indépendantes permet une grande modularité à la fois dans la définition des différentes applications et au sein même

des applications comme l'a illustré l'exemple de l'application IvP. Néanmoins, cela impose de définir une couche d'abstraction supplémentaire permettant de déterminer la structure de chaque application MOOS et la manière dont celles-ci interagissent ensemble.

2.4.2 Architectures de contrôle

Les architectures de contrôle robotique sont au cœur du développement des logiciels de contrôle puisqu'elles permettent de les structurer et ainsi de faciliter leur développement. Elles intègrent donc souvent les mécanismes d'implémentation issus des *Middlewares* et répartissent les différentes entités logicielles que ceux-ci proposent en fonction de leur contribution au logiciel de contrôle.

L'architecture CLARATy [VNE+00] est une architecture de contrôle basée objet. En effet, les objets et certains concepts associés tels que l'héritage permettent de considérer le système robotique à différents niveaux d'abstraction du plus générique au plus spécifique à une technologie donnée. Une vue générale de celle-ci est proposée Figure 2.19.

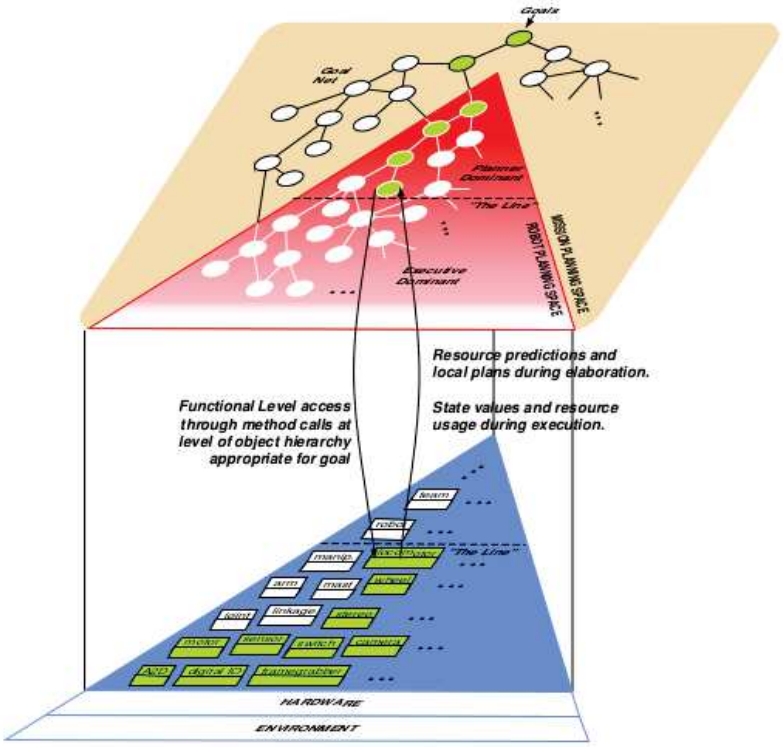

FIGURE 2.19 – L'architecture CLARATy (figure tirée de [VNE+00, Figure 1.6])

Il s'agit d'une architecture hybride [PACGD14] structurée en deux couches, la couche décisionnelle et la couche fonctionnelle. La couche décisionnelle est chargée de la prise de décision afin d'assurer le bon déroulement de la mission et du pilotage de l'exécution de la couche

fonctionnelle. Elle est conçue autour d'un réseau hiérarchisé d'objectifs qui contient les informations relatives à l'ordonnancement des différents objectifs en fonction de leurs contraintes temporelles et de précédence.

La couche fonctionnelle comprend les différentes fonctionnalités robotiques pouvant être mises en œuvre. Les différentes entités la constituant sont décrites suivant le paradigme objet qui permet une description modulaire du contrôleur robotique et la manipulation des différentes fonctionnalités à différents niveaux d'abstraction. Ainsi, un ensemble de classe abstraites sont proposées pour représenter les fonctionnalités génériques d'un composant puis sont raffinées en objets dont ceux du niveau d'abstraction le moins élevé décrivent l'implémentation de la fonctionnalité de manière spécifique au matériel utilisé. Ainsi, les classes les plus abstraites fournissent à la couche décisionnelle une interface générique lui permettant de commander les fonctionnalités de la couche fonctionnelle (par exemple faire avancer le robot à une vitesse choisie) sans se soucier de leur implémentation précise qui est prise en charge par les classes les moins abstraites (par exemple, faire tourner les roues ou les hélices à telle vitesse).

Ainsi, il est posssible de distinguer trois types principaux de classes dans l'architecture CLARATy :

- Les *Data Structure Classes* représentent les types de données échangées.
- Les *Generic Physical Classes* décrivent la représentation générique des différentes fonctionnalités physiques du robot (capteurs, actionneurs notamment) et elles doivent être spécialisées en fonction du matériel utilisé.
- Les *Generic Functional Classes* représentent les algorithmes de contrôle génériques applicables au robot. Similairement aux classes physiques, elles peuvent être spécialisées pour les adapter à un vecteur robotique particulier.

Néanmoins cette architecture présente certaines limitations. Ainsi même si l'approche objet permet une grande souplesse dans la décomposition des différentes entités et lois de commande, la problématique de la modularisation de ces dernières et notamment des connaissances qu'elles impliquent n'est pas vraiment abordée. En outre, même si la couche décisionnelle permet de prendre en charge certaines questions temporelles, les aspects temporels sont évoqués en termes de précédence et de durée mais les aspects liés à la récurrence des calculs ne sont pas mentionnés. En outre, les aspects temporels de l'exécution ne sont pas mis en relation avec certaines propriétés du système telles que la stabilité.

Nous devons également mentionner l'approche Chimera [Ste01] qui se base sur des entités appelées *Port Based Objects* qui permettent aussi de décrire le logiciel de contrôle à différents niveaux d'abstraction. Ces objets sont regroupés en assemblages appelés *Jobs* et qui portent des propriétés temporelles (période d'exécution, temps maximal d'exécution) permettant la

validation de l'ordonnancement. Néanmoins, il manque là aussi une approche pour articuler ces travaux avec les préoccupations des automaticiens telles que la stabillité.

Une autre architecture de contrôle intéressante est l'architecture développée par le LAAS [ACF+98], [ILLCP07]. Une vue générale de cette architecture, qui est décomposée en 3 couches, est donnée Figure 2.20. Les différentes fonctionnalités robotiques sont structurées de manière modulaire en entités logicielles appelées modules et qui sont utilisées dans la couche fonctionnelle.

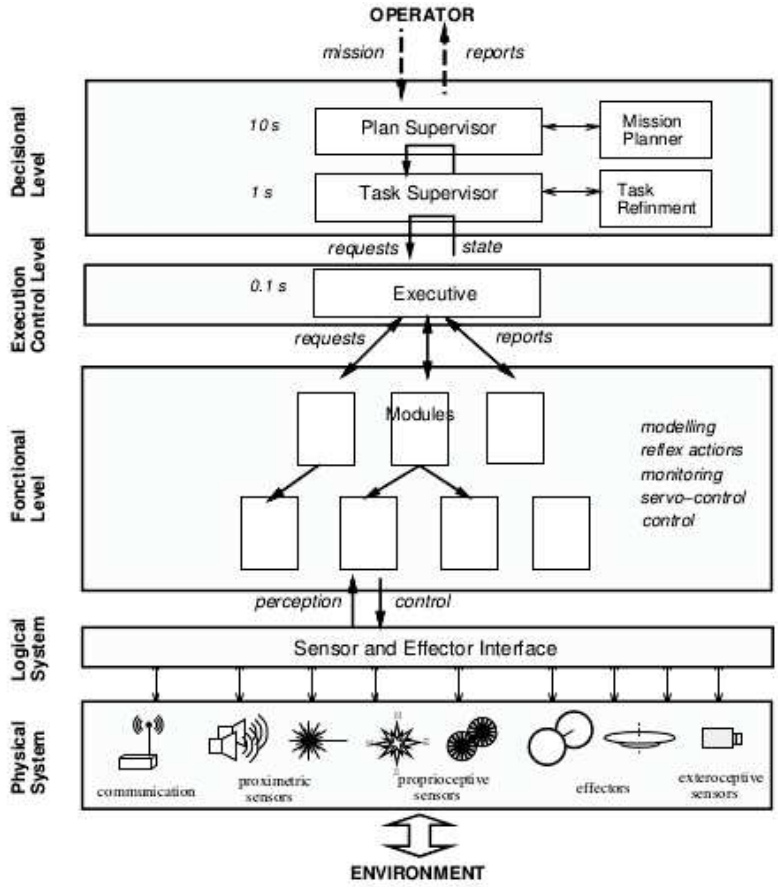

FIGURE 2.20 – L'architecture proposée par le LAAS (figure tirée de [ACF+98, Figure 1])

La couche décisionnelle est la couche chargée de superviser le déroulement de la mission et de prendre les décisions qui nécessitent une connaissance globale des objectifs de mission et du contexte d'exécution du robot et de planifier en conséquence les actions à entreprendre.

La couche de contrôle d'exécution est chargée de réaliser l'interface entre les couches fonctionnelles et décisionnelles. Elle permet notamment de combler la différence entre les dynamiques d'exécution des lois de commande qui doivent respecter la dynamique du vecteur robotique et celles du processus décisionnel qui peut mettre plusieurs secondes à réagir.

La couche fonctionnelle est celle où sont implémentées les différentes fonctionnalités du

robot (asservissements, lectures capteurs, écritures actionneurs). Celles-ci sont encapsulées dans des entités logicielles appelées modules. La couche fonctionnelle est ainsi décrite comme un réseau modulaire appelé *graphe d'activités* comme illustré Figure 2.21. La constitution de ce graphe est réalisée de manière dynamique par la couche exécutive en fonction des ordres que celle-ci reçoit de la couche décisionnelle.

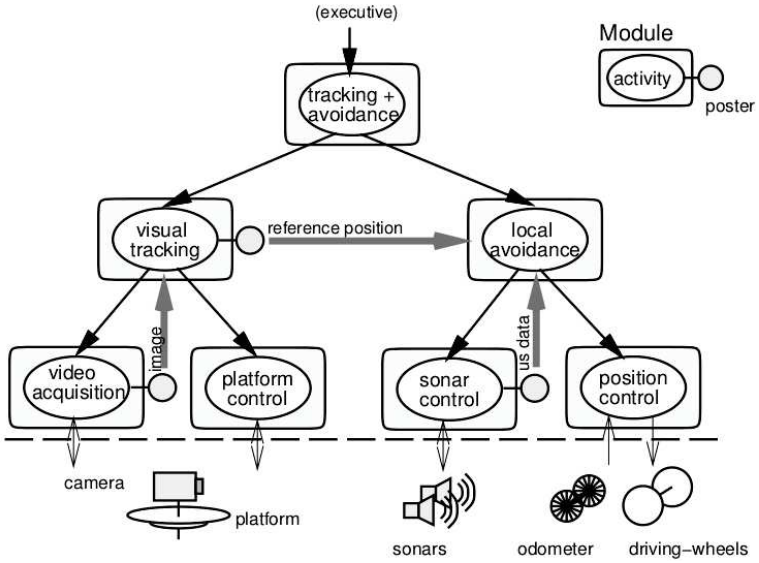

FIGURE 2.21 – Exemple de graphe d'activités dans l'architecture du LAAS (figure tirée de [ACF+98, Figure 2])

Les modules de cette architecture sont développés à l'aide du *framework* $G^{en}OM$ [FHC97], [CSH+11]. Une représentation schématique d'un module est proposée Figure 2.22.

Un module est chargé d'encapsuler une fonctionnalité robotique (il peut s'agir d'un algorithme ou de la gestion d'un capteur par exemple). Chaque module offre un ensemble de *services* pouvant être démarrés, interrompus ou paramétrés via des *requêtes*. Plusieurs *services* peuvent être exécutés simultanément. Un service en cours d'exécution est appelé *activité*.

Afin de lier le code utilisateur aux différents *services*, celui-ci doit être décomposé en entités calculatoires élémentaires appelées *codels*. Ceux-ci correspondent aux différentes parties du code utilisateur (initialisation, corps, terminaison). Ainsi, une *activité* correspond à l'exécution d'un *codel* dans une *tâche d'exécution* dont le fonctionnement va dépendre des contraintes temporelles associées à l'*activité*. Celle-ci peut être soit apériodique soit périodique auquel cas l'utilisateur devra spécifier une période d'exécution. Il est à noter que des *activités* s'exécutant à la même période au sein du même module pourront être regroupées dans la même *tâche d'exécution*. Enfin, les modules communiquent entre eux via des structures de données spécifiques appelées *posters*.

Récemment, cette approche a été modifiée en remplaçant $G^{en}OM$ par le *framework* BIP

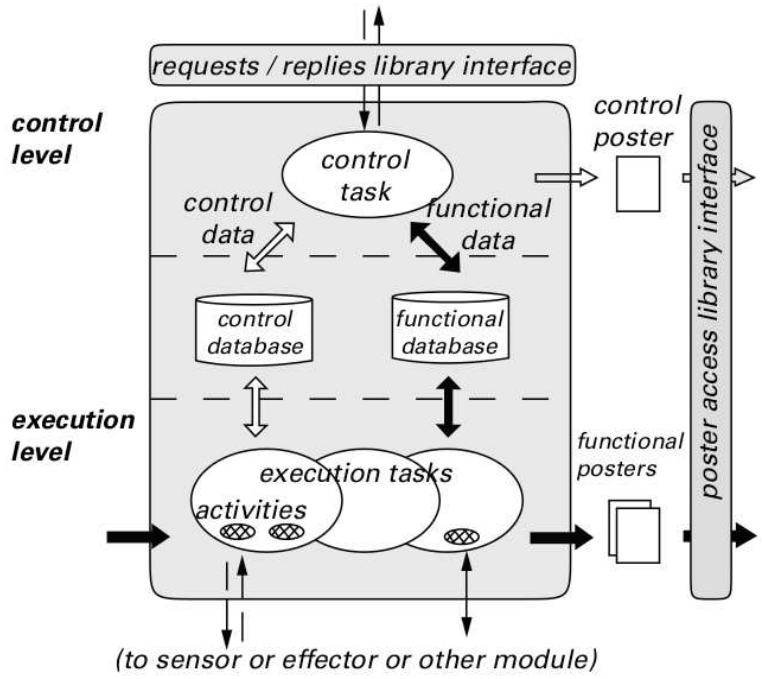

FIGURE 2.22 – Modèle de module $G^{en}OM$ (figure tirée de [PACGD14, Figure 11])

[BGL+08] permettant de modéliser des composants temps-réel hétérogènes. BIP propose un modèle de composants atomiques (i.e. minimaux) avec trois aspects, un comportement, des interactions et des priorités entre celles-ci. Ainsi la couche fonctionnelle peut être décrite comme un assemblage hiérarchisé de tels composants. Cela permet de vérifier un certain nombre de propriétés de l'architecture comme le respect d'un certain nombre de propriétés temporelles (par exemple que le temps mis entre la détection d'un obstacle et la réaction à celle-ci soit inférieur à une certaine durée) ainsi que logiques, notamment l'absence de blocage mortel dans l'application.

Comme nous l'avons vu l'approche orientée module proposée par $G^{en}OM$ puis par BIP permet une grande modularité dans la définition des applications robotiques. La décomposition d'une fonctionnalité en *codels* souligne le besoin de proposer une description modulaire avec un fin grain de décomposition des lois de commande afin de pouvoir s'articuler plus efficacement avec des approches proposant une telle structuration. Il faut également noter que si $G^{en}OM$ impose la détermination d'une période d'exécution pour les différentes *activités* périodiques, il manque un cadre formel permettant de relier celles-ci aux préoccupations des automaticiens telles que la stabilité.

L'approche ORCCAD (*Open Robot Controller Computer Aided Design*) [SPGA06] est également très intéressante car elle essaie de permettre une meilleure liaison avec les automati-

ciens. Elle a été conçue afin de se focaliser sur les aspects de validation logico-temporelle de l'exécution du logiciel de contrôle. Elle propose un modèle d'architecture conçu selon deux couches, la couche commande et la couche application comme représenté Figure 2.23.

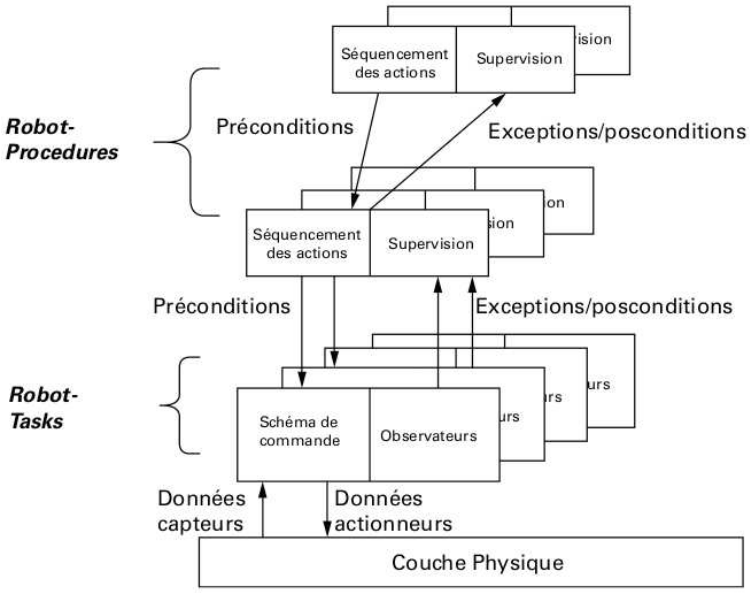

FIGURE 2.23 – L'architecture ORCCAD (figure tirée de [PACGD14, Figure 13])

La couche commande est décrite comme un ensemble de *Robot-Tasks*. Celle-ci représente la mise en œuvre d'un schéma de commande et d'un ensemble d'observateurs d'état associés. Les outils utilisés dans ORCCAD permettent de décrire les *Robot-Tasks* comme un assemblage modulaire de composants préexistants appelés *Module-Tasks*. De fait, si une *Robot-Task* est destinée à être le niveau de description minimal d'ORCCAD, il s'agit du niveau maximal de description pour un automaticien [EKJ96].

En outre, il est nécessaire de spécifier un ensemble d'informations permettant de paramétrer le comportement temporel des différents *Module-Tasks* contenus dans la *Robot-Task*. Ceux-ci comprennent les périodes d'exécution, les pires temps d'exécution estimés (*Worst Case Execution Time*, WCET) ainsi que les relations décrivant les synchronisations entre modules. Deux modules peuvent également être temporellement couplés et fonctionner à la même période et enfin l'exécution d'un module peut être synchronisée sur un signal ou une donnée capteur. Ces propriétés temporelles permettent de vérifier la cohérence temporelle de l'application [EKJ96].

La couche application est basée sur des entités appelées *Robot-Procedures*. Celles-ci permettent de décrire de manière hiérarchique des actions à accomplir jusqu'à la description de la mission elle-même. En effet, une *Robot-Procedure* représente un assemblage de *Robot-Tasks* voire de *Robot-Procedures*. Elle décrit également le processus décisionnel permettant de sélectionner les constituants à utiliser [SPGA06]. La description d'une *Robot-Procedure* dans le

langage ESTEREL permet de vérifier le comportement logique de l'application (notamment qu'il n'y ait pas de *deadlock* dans l'application décrite). Elle permet en outre de s'assurer du déterminisme de l'application ce qui est critique afin de s'assurer de son bon fonctionnement et de sa reproductibilité.

Une des principales limitations de cette approche est le niveau d'abstraction assez élevé des *Module-Tasks* qui ne font ainsi pas ressortir les contributions des différents domaines scientifiques impliqués dans la conception d'un système robotique.

Enfin, l'approche proposée dans [RM10a] et [RM10b], s'il ne s'agit pas d'une architecture à proprement parler, doit être mentionnée pour son intérêt dans la vérification des propriétés temporelles d'une architecture logicielle. En effet, elle se base sur des composants logiciels (considérés comme des processus sensori-moteur) décrits comme un assemblage d'entités élémentaires appelées *codels*. Sur ces derniers, différentes contraintes sont répercutées : leur pire temps d'exécution (WCET), leurs fréquences minimales et maximales de fonctionnement, les ressources (notamment matérielles) utilisées par les *codels* afin de détecter leurs exclusions mutuelles (i.e. l'impossibilité de s'exécuter de manière simultanée) au moment de l'exécution ainsi que leurs relations de communication.

A partir de là, un certain nombre d'opérateurs de composition sont définis pour diffuser les contraintes portant sur les codels à leur composition. Ainsi, cela permet de vérifier les propriétés de cette composition et de produire automatiquement un ordonnancement compatible avec les différentes contraintes. En outre, il faut noter que les opérateurs de composition peuvent être définis à diférents niveaux d'abstraction (opérateurs génériques à opérateurs dépendant du contexte de déploiement, par exemple du nombre de processeurs disponibles, des entités logicielles).

Néanmoins, cette approche présente certaines limitations. Elle n'aborde pas la conception des *codels* qui restent des entités très abstraites et ne sont pas reliées aux différentes connaissances impliquées dans le développement d'une architecture de contrôle d'un robot. Elle doit également être complétée afin de relier de manière plus formelle les différentes contraintes temporelles aux problématiques des automaticiens. Enfin, la problématique d'utilisation des contraintes afin de faciliter l'identification des points bloquants en cas d'impossibilité de trouver un ordonnancement compatible n'est pas évoquée.

2.4.3 Approches avec lien entre automatique et génie logiciel

La grande majorité de ces approches ne prend pas réellement en compte la problématique de l'implémentation des lois de commande et les aspects temporels qui y sont associés et qui

représentent, comme nous l'avons vu précédemment, une préoccupation importante pour les automaticiens. La raison principale de ce manque d'interaction réside dans le fait que la représentation de ces aspects pour les automaticiens d'un côté et les ingénieurs logiciels de l'autre est différente, ce qui rend très difficile les échanges entre ces deux domaines. Il manque donc un cadre commun permettant d'articuler les travaux et les approches de ces deux communautés, qui se retrouvent autour des systèmes cyber-physiques et de l'*autonomic computing*, par exemple.

Dans [UFH+12], une architecture de contrôle est proposée afin de réaliser un point d'interaction entre ces deux domaines permettant de découpler les préoccupations de chaque acteur du développement. Cette approche se focalise essentiellement sur les aspects d'implémentation liés au fonctionnement des différents capteurs et actionneurs du système. En effet, dans une conception monolithique les spécificités et performances (période de mise à jour, incertitude sur les données mesurées) du matériel utilisé sont directement intégrées à la conception de la loi de commande afin d'obtenir les performances désirées. Dès lors, un changement technologique (remplacement d'un capteur par un autre avec des performances différentes) a un impact majeur sur les propriétés de l'architecture de contrôle et les performances du contrôleur mis en œuvre.

Cette approche s'inspire donc des travaux traitant de la fusion de données issues de capteurs pour proposer d'intercaler des couches d'abstraction entre les *drivers* des composants matériels et les différents composants logiciels chargés du contrôle du robot. Cette approche aboutit à l'architecture logicielle présentée Figure 2.24.

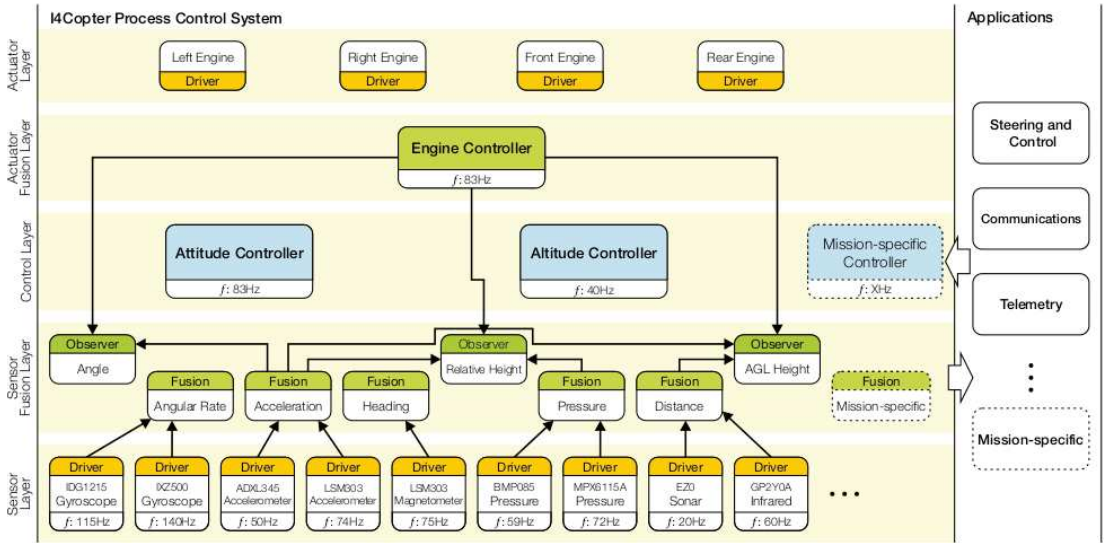

FIGURE 2.24 – Architecture logicielle permettant de découpler contrôle et *drivers* (figure tirée de [UFH+12, Figure 2])

Cette architecture utilise différents composants logiciels. La description de l'interface de ces différents composants logiciels est définie par quatre informations. Elle contient l'unité et la valeur du mesurande ainsi que l'intervalle de valeurs valides pour celui-ci. Enfin, l'incertitude de mesure est le dernier élément utilisé dans la description de l'interface car le bruit capteur a un impact important sur les performances du contrôle. Dans cette approche, l'incertitude de mesure est considérée comme l'élément commun permettant l'interaction entre automatique et conception de logiciel temps-réel. Ainsi, dans l'architecture présentée dans la Figure 2.24, les différentes stratégies de commande sont mises en œuvre dans la couche de contrôle (*Control Layer*). L'automaticien va fixer les entrées/sorties du composant, l'incertitude maximale tolérée pour chacune d'entre elles ainsi que la période de fonctionnement de chaque composant logiciel chargé du contrôle afin d'atteindre les performances désirées.

Les couches de fusion (*Sensor Fusion Layer* et *Actuator Fusion Layer*) sont chargées de fournir les données en respectant les contraintes fixées par les automaticiens créant une couche d'abstraction entre les entités implémentant le contrôle et les différents *drivers* du système. La gestion de ces couches assure également l'adaptation des différentes stratégies de fusion lors de l'exécution. Ainsi, si l'utilisation d'une stratégie de fusion ou d'un capteur ne permet plus de satisfaire les besoins du contrôle ou que la stratégie de contrôle change, alors les différentes entités logicielles utilisées dans les couches de fusion et les capteurs mobilisés seront adaptés en conséquence par l'architecture.

Ainsi cette approche propose une séparation des préoccupations entre les deux domaines. Dès lors, chacun peut poursuivre ses propres développements sans tenir compte des problématiques de l'autre, l'articulation étant assurée par l'architecture proposée. Nous pensons qu'il s'agit là d'une limitation à ces travaux car l'absence d'interaction entre les deux domaines masque aux concepteurs certains phénomènes qui influencent le comportement du robot (notamment sa stabilité). Le choix des périodes d'exécution est donc laissé à la seule appréciation des automaticiens en fonction de leurs préoccupations mais d'autres éléments tels que les performances du calculateur ont également une grande influence sur le choix des périodes d'implémentation et ne peuvent être pris en compte avec cette approche. En outre, la problématique de la conception des entités de la couche de contrôle et notamment d'un effort vers une plus grande modularité de celles-ci n'est pas évoquée dans le cadre de ces travaux.

A la Section 2.2.1, nous avons présenté les systèmes hybrides qui combinent évolution continue et logique discrète. Leur complexité couplée à une importance croissante de l'utilisation d'un tel modèle (qui correspond à tout système couplant des composants "réels" et informatiques comme un robot) ont poussé au développement de l'approche des Automates Hybrides

[Hen00]. Un système hybride est ainsi représenté comme un graphe dont les modes de contrôle sont les noeuds et les transitions les arcs. La Figure 2.25 présente un exemple trivial d'un tel graphe.

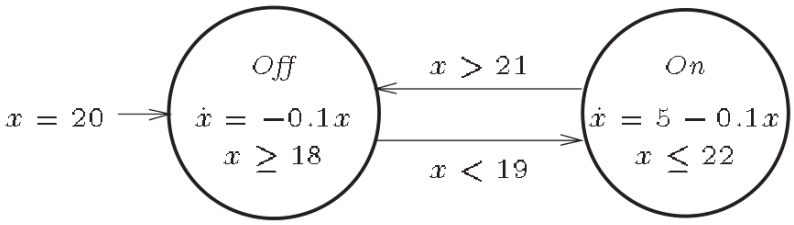

FIGURE 2.25 – L'automate hybride associé au contrôle d'un thermostat (figure tirée de [Hen00, Figure 1])

Ainsi, un tel automate possède un certain nombre de variables internes $X = \{x_1, ...x_n\}$. Dans l'exemple de la Figure 2.25, la seule variable interne est la température x. Sur chaque transition, un prédicat portant sur tout ou partie de X, appelé condition de saut, est posé et décrit les conditions dictant les changements de mode (par exemple le thermostat s'éteint si la température dépasse 21 degrés). Dans chaque mode de contrôle, un prédicat, appelé condition de flux (*flow condition*), décrit l'évolution continue de l'état du système (par exemple $\dot{x} = 5 - 0.1x$ si le thermostat est en mode *On*) et une condition invariante que doit respecter l'état du système est aussi posée (par exemple, le thermostat doit s'éteindre au plus tard si la température atteint 22 degrés). La combinaison des conditions de saut et des invariants d'état permet qui plus est de faire ressortir certains aspects temporels du système comme sa réactivité. Dans notre exemple, le thermostat doit commencer à se couper (passage en mode *Off*) si la température dépasse 21 degrés mais cette dernière ne doit pas dépasser 22 degrés. Afin de garantir le respect de ces prédicats, le temps de transition pour passer de l'état *On* à l'état *Off* est donc limité. La durée maximale de commutation peut être estimée à l'aide du prédicat de flux de l'état *On*.

Les systèmes plus complexes peuvent être représentés par une composition d'automates hybrides. Le formalisme permet de vérifier si la composition de tels automates est réalisable [LSV03] et, si c'est le cas, d'étudier les propriétés du système résultant de cette composition. Enfin, les travaux proposés dans [KLSV03] limitent les interactions entre les automates à des évènements discrets (alors que les automates hybrides présentés dans [LSV03] et [Hen00] peuvent partager des variables d'état internes ce qui entraîne une influence mutuelle continue entre les automates). Cela facilite ainsi l'étude des propriétés de tels automates tout en conservant leur capacité de représentation d'un système hybride.

Ainsi cette approche permet une grande modularité dans la description d'un système de nature hybride tout en apportant un formalisme d'étude des compositions de tels systèmes.

En outre, cette approche met en avant certains aspects temporels liés au système (notamment sur sa réactivité). Néanmoins, les différentes contributions au comportement du système ne sont pas réifiées. Par exemple, dans l'automate de la Figure 2.25, on voit que dans le mode *On*, l'effet de l'environnement (le terme $-0.1x$) n'est pas dissocié de l'effet du thermostat lui-même (le terme 5). En outre, les questions de l'implémentation des automates et de l'impact des propriétés temporelles qui ressortent de leur analyse sur l'implémentation ne sont pas abordées.

La conception de systèmes cyber-physiques (CPS) dont font partie les robots implique de nombreux acteurs comme nous l'avons présenté lors du chapitre précédent. L'hétérogénéité des méthodes utilisées dans chaque discipline nécessite de combler l'écart théorique et sémantique entre les différents intervenants. Partant de ce postulat, une approche est proposée dans [DLTT13] pour construire un cadre d'interaction entre les domaines de l'automatique et de la conception logicielle. En effet, chaque choix de conception d'un domaine ayant un impact sur l'autre, il est nécessaire de réifier explicitement toutes les interconnexions entre ces deux disciplines afin de permettre aux concepteurs de déterminer les approches qui satisferont les contraintes et les besoins de chacune. Pour se faire, [DLTT13] propose d'établir des contrats de conception rendant explicites les attentes et les devoirs de chaque partie. L'approche par contrat comprend deux points essentiels, la définition des différents contrats possibles et un ensemble de règles permettant de guider le choix du contrat adapté à la situation.

Dans le cadre des travaux proposés, ce sont les aspects de contraintes temporelles qui sont traités. Soit un contrôleur M (dans [DLTT13], il est supposé être une machine de Mealy, mais cette approche peut être étendue). Celui-ci, en plus du calcul du contrôle proprement dit (i.e. calcul des sorties et du nouvel état interne en fonction des entrées et de l'état interne courant du contrôleur), possède une fonction permettant de lier les données capteurs aux entrées proprement dites du contrôleur et une seconde fonction chargée de convertir ses sorties en consignes actionneurs. Les variables temporelles à prendre en compte sont représentées dans la Figure 2.26.

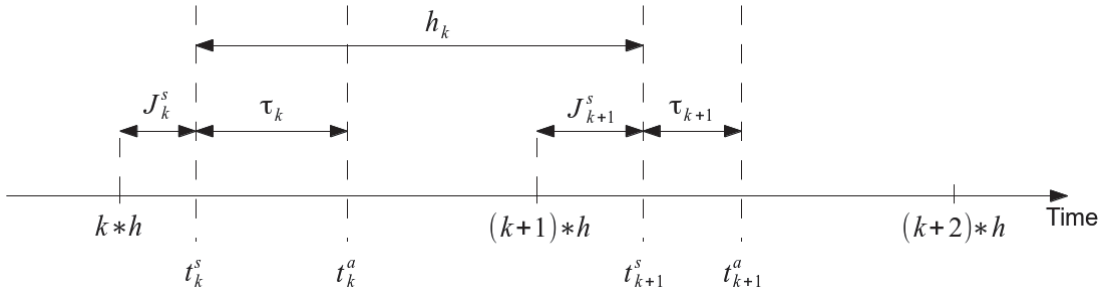

FIGURE 2.26 – Variables temporelles impliquées dans un contrat de conception (figure tirée de [DLTT13, Figure 2])

Les différentes variables impliquées sont définies comme :

- $k \in \mathbb{N}^*$ indique le cycle d'exécution
- h : période d'exécution théorique du contrôleur, $h * k$ désigne donc la date théorique de début du k-ième cycle d'exécution.
- t_k^s : date de lecture des capteurs.
- t_k^a : date d'envoi des commandes aux actionneurs.
- $\tau_k = t_k^a - t_k^s$: délai entre la lecture capteur et l'envoi des commandes aux actionneurs (délai StA, *sampling-to-actuation*).
- $J_k^s = t_k^s - k * h$: décalage entre la date de lecture des capteurs et le début théorique du cycle d'exécution.
- $h_k = t_{k+1}^s - t_k^s$: "période" d'exécution effective, c'est-à-dire délai réel entre le début de deux cycles consécutifs. Il est à noter que nous avons placé le terme période entre guillemets car, sous l'hypothèse de décalages variables dans le temps, h_k est susceptible de varier d'un cycle d'exécution à l'autre.

A partir de ces variables, quatre contrats sont proposés, chacun correspondant à des hypothèses de conception différentes :

- *Zero Execution Time* (ZET) : le contrat est défini comme le couple (M, h). Il impose $J_k^s = \tau_k = 0$, c'est-à-dire qu'il n'y ait aucun délai dans l'architecture et que les calculs soient effectués de manière instantanée. Il s'agit là bien entendu d'un contrat décrivant un cas idéal.
- *Deadline Execution Time* (DET) : le contrat est défini comme le triplet (M, h, d). d est une *deadline* telle que $d < h$. Les calculs doivent être effectués sans délai et avant la fin de cette date limite ce qui signifie que $J_k^s = 0$ et $\tau_k \leq d$.
- *Logical Execution Time* (LET) : le contrat est défini comme le couple (M, h). Il impose que $J_k^s = 0$ et que les commandes soient envoyées aux actionneurs à la fin de la période, c'est-à-dire $t_k^a = t_{k+1}^s = (k+1) * h$ et donc $\tau_k = h$.
- *Timing Tolerances* (TOL) : il est défini comme le 5-uplet $(M, h, \tau, J^h, J^\tau)$. h et τ sont les valeurs nominales de la période et du délai StA respectivement. J^h et J^τ définissent les variations admissibles de ces deux paramètres. Cela signifie qu'à chaque cycle les inégalités $h_k - h \leq J^h$ et $\tau_k - \tau \leq J^\tau$ doivent être respectées. En outre, le contrat impose que $J^\tau \leq \tau$ et $J^h + \tau + J^\tau < h$.

Une fois les contrats définis, pour chaque application, les acteurs impliqués doivent s'accorder sur le contrat à sélectionner et fixer les valeurs des différents paramètres (par exemple, les valeurs de M, h et d pour un contrat DET). Cela implique notamment de s'intéresser à la fois à la tolérance du contrôle aux délais induits (ce qui impacte sa performance : stabilité,

précision par exemple) mais également aux performances de la cible d'exécution. La bonne adéquation de ce contrat est très importante car, en cas de violation du contrat, cela peut dégrader les performances du contrôle (causant par exemple une instabilité). Le processus de conception et l'apport des contrats sont présentés Figure 2.27.

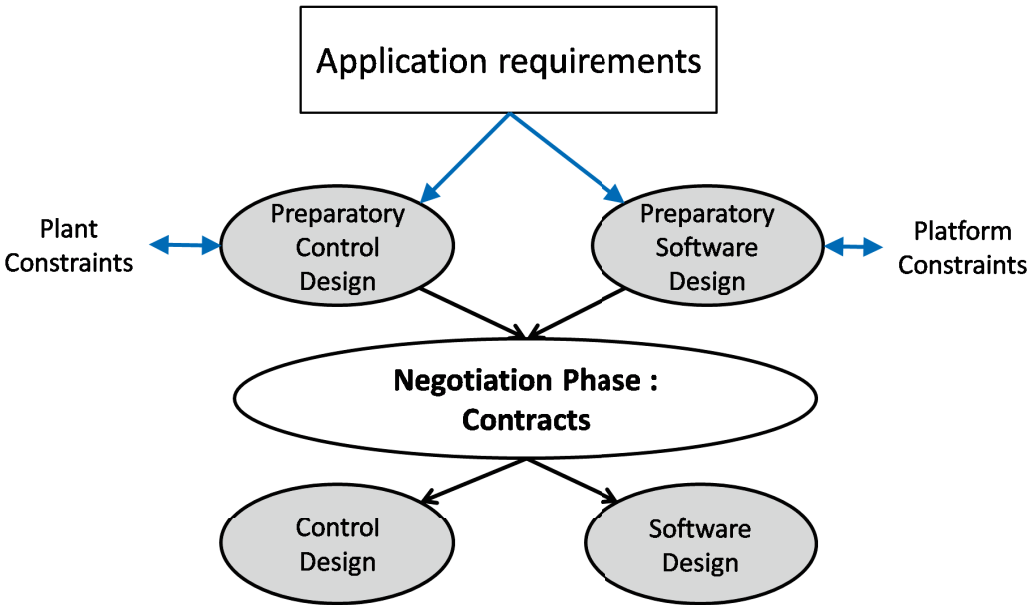

FIGURE 2.27 – Utilisation des contrats comme interface entre les processus de conception de l'automatique et du génie logiciel

Dans l'approche proposée, la conception se déroule en deux phases. Dans la première, chaque acteur va développer individuellement des solutions pour l'application traitée en fonction de contraintes spécifiques à chacun (flèches bleues). Ensuite la phase de négociations va permettre de déterminer les solutions retenues et les modifications à y apporter par chaque partie prenante, ce qui va leur permettre, chacune de leur côté, de finaliser les solutions à mettre en œuvre.

Néanmoins cette approche présente la limitation, à nos yeux, de se focaliser essentiellement sur l'aspect de l'interaction entre ces deux domaines et de ne pas proposer une reformulation de certaines approches de conception afin de favoriser leur interaction. Ainsi, repenser les approches de conception afin de placer au centre du cadre commun des préoccupations telles que la modularité (cela vaut surtout pour l'automatique) ou la stabilité permettrait une meilleure interaction tout au long du processus de conception. De plus, cette approche, qui suggère de mettre en place des contrats entre tous les acteurs du processus de développement, ne considère pas le cas conflictuel où diverses négociations imposeraient des termes contradictoires à une partie prenante.

2.5 Points clés du chapitre

- Le contrôleur d'un robot est tout d'abord décrit sous forme de lois de commande avant d'être implémenté sous forme de logiciel de contrôle.
- Le logiciel de contrôle est structuré suivant une architecture logicielle mettant en œuvre des mécanismes temps-réel.
- La stabilité est une considération essentielle pour les automaticiens.
- La stabilité d'un contrôleur est souvent étudiée en temps continu en utilisant principalement la théorie de Lyapunov.
- Lorsque le contrôleur s'exécute en temps discret, la notion de stabilité est augmentée par un ensemble de périodes d'exécution pour lesquelles celle-ci est garantie.
- Les outils pour déterminer ces périodes sont peu nombreux et lourds à mettre en œuvre.
- La prise en compte de la problématique de modularité demeure limitée en automatique et le lien avec l'implémentation du contrôle est rarement fait.
- Le langage de modélisation Modelica propose une description des systèmes permettant de mettre en lumière leurs différents composants, leur rôle et les interactions entre eux.
- Les architectures logicielles et les *Middleware* accordent une grande place à la modularité. Cependant ils ne prennent pas en compte les aspects liés à la conception du contrôle et relativement peu ceux liés à la stabilité de celui-ci.
- Certaines approches cherchent pourtant à établir un pont entre automatique et logiciel de contrôle mais sont souvent limitées car elles visent à séparer les préoccupations de ces deux domaines en travaillant à un niveau plus élevé d'abstraction.
- Les contrats de conception proposent un cadre d'interaction entre automatique et génie logiciel au niveau des considérations temporelles mais ne considèrent pas la question de la conception du contrôle proprement dite.
- L'approche ORCCAD souligne le fait que la vérification des propriétés temporelles de l'application est essentielle afin de s'assurer du bon fonctionnement de l'application robotique.

Chapitre 3

Positionnement

L'un de nos principaux objectifs est de parvenir à explorer des aquifères karstiques afin de récolter les données permettant aux scientifiques qui étudient ces milieux de mieux les comprendre. Comme nous l'avons expliqué au Chapitre 1, il s'agit de milieux d'une très grande complexité et leur exploration robotisée représente un véritable défi scientifique. Du point de vue de la commande du robot, il n'est pas concevable de laisser celui-ci au seul contrôle d'un opérateur qui aurait de grandes difficultés à le manœuvrer de manière sûre dans un tel environnement. Qui plus est, une perte de communication filaire (qui peut soit arriver parce que le câble a été endommagé par les parois du conduit, soit être volontairement déclenchée par l'opérateur ou le robot si l'ombilical venait à se coincer) engendrerait alors soit la perte inéluctable du vecteur robotique et de son coûteux équipement soit l'activation d'un mode de fonctionnement autonome. Il est donc nécessaire de doter le robot d'une grande autonomie opérationnelle afin qu'il puisse faire face aux situations qu'il peut rencontrer lors de l'exploration. De par la spécificité de ces environnements, il est, en outre, presque indispensable d'intégrer l'expertise des spécialistes du milieu (hydrogéologues, plongeurs) pour aider les automaticiens à concevoir l'architecture de contrôle du robot à la fois pour choisir les stratégies de contrôle les mieux adaptées à de tels environnements mais aussi pour proposer des modèles qui peuvent être utilisés dans le contrôle du robot. Cette expertise est de plus amenée à évoluer au fil des expérimentations et des données récoltées. Enfin, la complexité de tels milieux impose une validation progressive de nos choix et approches ce qui nécessite de réaliser des expérimentations dans des milieux qui laissent une marge d'erreur plus grande. Ces milieux ont néanmoins leurs propres spécificités et imposent donc de mettre en œuvre des modèles ou des lois de commande spécifiques en supplément ou en lieu et place de ceux utilisés pour l'exploration d'aquifères karstiques. Il nous faut donc pouvoir discerner les modèles et lois de commande transversaux entre différentes applications robotiques afin de pouvoir les réutiliser après qu'ils aient été validés, de ceux qui sont spécifiques à l'application et évolueront donc d'une application à l'autre.

Dans ce contexte, nous pensons que répondre à un tel défi scientifique impose de repenser la manière de concevoir l'architecture de contrôle d'un robot. En effet, le paradigme traditionnel qui conduit à une conception monolithique de la commande pénalise fortement l'intégration de connaissances provenant de divers domaines scientifiques puisque celles-ci doivent être maîtrisées et comprises par l'automaticien s'il veut pouvoir en faire une utilisation efficace. La conception monolithique de la commande empêche également de réutiliser efficacement des éléments de l'architecture de contrôle d'une application robotique à une autre.

Pour cela, nous pensons qu'il est désormais nécessaire de structurer la description des architectures de contrôle autour des *connaissances*. Cela passe par la définition d'un nouveau formalisme de représentation centré sur la notion de connaissance. Ce formalisme doit permettre d'expliciter la contribution de chaque connaissance ainsi que les interactions entre elles, ce qui va naturellement structurer les architectures de contrôle, favoriser la réutilisation et permettre un meilleur échange d'expertise entre les différents acteurs du développement du robot. Ces travaux s'inscrivent dans une approche relevant de l'ingénierie dirigée par les modèles.

Pour ce faire, le formalisme doit permettre de décrire un contrôle comme un assemblage de connaissances. Mais comme ces dernières sont hétérogènes et susceptibles d'évoluer dans le temps, il faut pouvoir les manipuler de manière uniforme et maîtriser l'impact de leurs évolutions sur la conception des lois de commande. Dès lors, il faudrait pouvoir encapsuler les connaissances dans des entités qui permettraient de les manipuler via une interface homogène. En ce sens, la définition de ces entités pourrait se rapprocher de ce qui est proposé dans une approche telle que Modelica. En effet, les approches orientées objet permettent de développer des entités avec une séparation claire entre leur code interne et l'interface permettant d'interagir avec d'autres entités. Néanmoins, ces entités doivent préserver le sens physique (i.e. ce qu'elles représentent concrètement) des connaissances.

Lorsque l'on parle de représentation orientée connaissance, certaines approches viennent naturellement à l'esprit telles que les ontologies, les *Domain Specific Languages* (DSLs) ou les *Architecture Description Languages* (ADLs). Les ontologies sont utilisées pour représenter et organiser les connaissances spécifiques à un domaine ainsi que les relations sémantiques et conceptuelles entre elles [Gru93]. De nombreuses ontologies ont été développées dans des domaines très divers allant de la robotique à la biologie en passant par l'économie. Les DSLs se rapprochent des ontologies dans le sens où ils servent à représenter les concepts et leurs relations dans un domaine scientifique mais sont généralement situés à un niveau d'abstraction moins élevé afin d'offrir des fonctionnalités telles que la génération automatique de code ou la vérification fonctionnelle [Wal09]. Les ADLs sont des langages utilisés pour permettre une représentation conceptuelle des composants logiciels et matériels d'une architecture et

des relations entre eux [Cle96]. Parmi les nombreuses approches proposées en robotique, nous pouvons ainsi citer Proteus [Far12]. Il s'agit d'une ontologie dont le but est de favoriser les échanges entre les différents acteurs du domaine robotique autour d'une sémantique commune. Ainsi les différentes parties prenantes peuvent utiliser l'ontologie pour décrire un problème. Un utilisateur conçoit une solution et la partage en incluant son champ d'application et d'autres utilisateurs peuvent en bénéficier et l'adapter à d'autres problèmes ou proposer leurs propres solutions. Le vocabulaire commun permet l'échange des solutions sans ambiguïté entre tous les utilisateurs de Proteus. Cette ontologie regroupe ainsi tous les concepts permettant de décrire toutes les entités impliquées dans une mission robotique, l'environnement, les divers agents (robots, intervenants humains), les algorithmes ou encore les objectifs de mission ainsi que les interactions entre ces différents éléments. Nous pouvons également souligner *Core Ontology for Robotic and Automation* (CORA) [PFC14] qui propose un effort de standardisation des concepts et des divers éléments sémantiques dans les domaines de la robotique et de l'automatique. Enfin, nous pouvons également mentionner le *Framework* SafeRobots [RMT14]. Celui-ci vise à représenter un problème de robotique suivant différents niveaux d'abstraction : la définition du problème (i.e. les objectifs à la fois en termes fonctionnels et non fonctionnels), la description des différentes solutions et l'espace opérationnel qui comprend tous les outils pour mettre en œuvre les solutions. La solution est décrite à un niveau d'abstraction élevé qui s'intéresse essentiellement aux interactions entre les différents constituants via la description d'un assemblage de composants appelés portes (délais, synchronisations, sélections, par exemple). Néanmoins, la description avec de telles entités fait perdre leur sens fonctionnel au profit d'une analyse des interactions au niveau système.

Le formalisme proposé ne va pas représenter les connaissances à un niveau d'abstraction aussi élevé que ces approches. En effet, le formalisme que nous devons proposer ne va pas manipuler de concepts mais des connaissances fonctionnelles, c'est-à-dire calculatoires (i.e. équations au sens large ou données exploitables dans un calcul). Dès lors, notre approche doit s'inscrire dans un cadre complémentaire des ontologies ou des DSLs car chaque approche va pouvoir capturer des connaissances qui sont très difficilement exprimables par l'autre approche.

Mais au-delà de la description d'une architecture de contrôle, il faut garder à l'esprit que celle-ci doit être mise en œuvre sous forme de logiciel de contrôle pour pourvoir effectivement être utilisée dans le cadre de la commande de robots. Ce logiciel de contrôle repose sur la structure et les mécanismes proposés par l'architecture logicielle et le *Middleware*. En outre, la cible technologique au sens large (calculateurs, capteurs, actionneurs) sur laquelle est implémenté le logiciel apporte ses propres contraintes qui vont impacter le comportement du

contrôle (par exemple, les périodes de mise à jour des données capteurs), dont par exemple la stabilité qui est critique. Des travaux comme [RM10a] et [RM10b] ont également souligné l'importance des contraintes temporelles dans la conception du logiciel de contrôle et la nécessité de les expliciter afin de pouvoir vérifier leur cohérence entre les différents composants logiciels utilisés.

Il est donc nécessaire d'intégrer à la fois les préoccupations des automaticiens et les contraintes technologiques de la cible d'implémentation comme souligné dans les travaux portant sur les *Design Contracts* [DLTT13]. Le formalisme proposé devra donc permettre d'exprimer des *Contraintes* portant sur les entités encapsulant les connaissances. De ce point de vue, il proposera une définition des entités plus riche que celle proposée dans des travaux tels que Modelica ou les MDLes [HVKA+00] où l'implémentation du contrôle n'est pas explicitement prise en compte. Ensuite, à des fins de vérification, ce formalisme devra également permettre d'exprimer la composition des *Contraintes* portant sur les différentes entités.

Sur les aspects implémentation, nous ne pensons pas qu'il soit nécessaire de proposer une nouvelle structuration du logiciel de contrôle. En effet, la littérature est plutôt vaste à ce sujet et de nombreuses approches basées sur les dernières avancées dans le domaine du génie logiciel existent. Nous pensons plutôt qu'il est nécessaire, pour pouvoir tirer un meilleur parti des solutions logicielles déjà existantes, d'accroître les relations, actuellement limitées par le manque d'un cadre commun, entre l'automatique et le génie logiciel. Notre approche doit donc permettre de capturer un ensemble de propriétés non exprimées dans les architectures logicielles traditionnelles. Néanmoins, contrairement à des travaux tels que [UFH+12] qui prônent une séparation des préoccupations, nous pensons qu'à l'instar de [DLTT13], il est nécessaire d'intégrer les préoccupations des deux domaines dans un cadre commun permettant une meilleure synergie entre eux.

De fait, il est nécessaire que notre approche se base sur une formalisation des contraintes qui aide à définir ce cadre commun pour proposer une méthodologie qui permet à tous les acteurs du développement du robot d'interagir et de les guider dans l'implémenation des lois de commande décrites avec ce formalisme.

Tout au long de ce manuscrit, nous allons donc détailler le formalisme proposé. Nous présenterons également la méthodologie que nous mettons en place à partir de ce formalisme afin de guider les ingénieurs logiciels vers l'utilisation des mécanismes les mieux adaptés d'un *Middleware* donné (ContrACT en l'occurrence) pour la mise en œuvre des entités logicielles implémentant les différentes connaissances. Celle-ci est représentée schématiquement Figure 3.1.

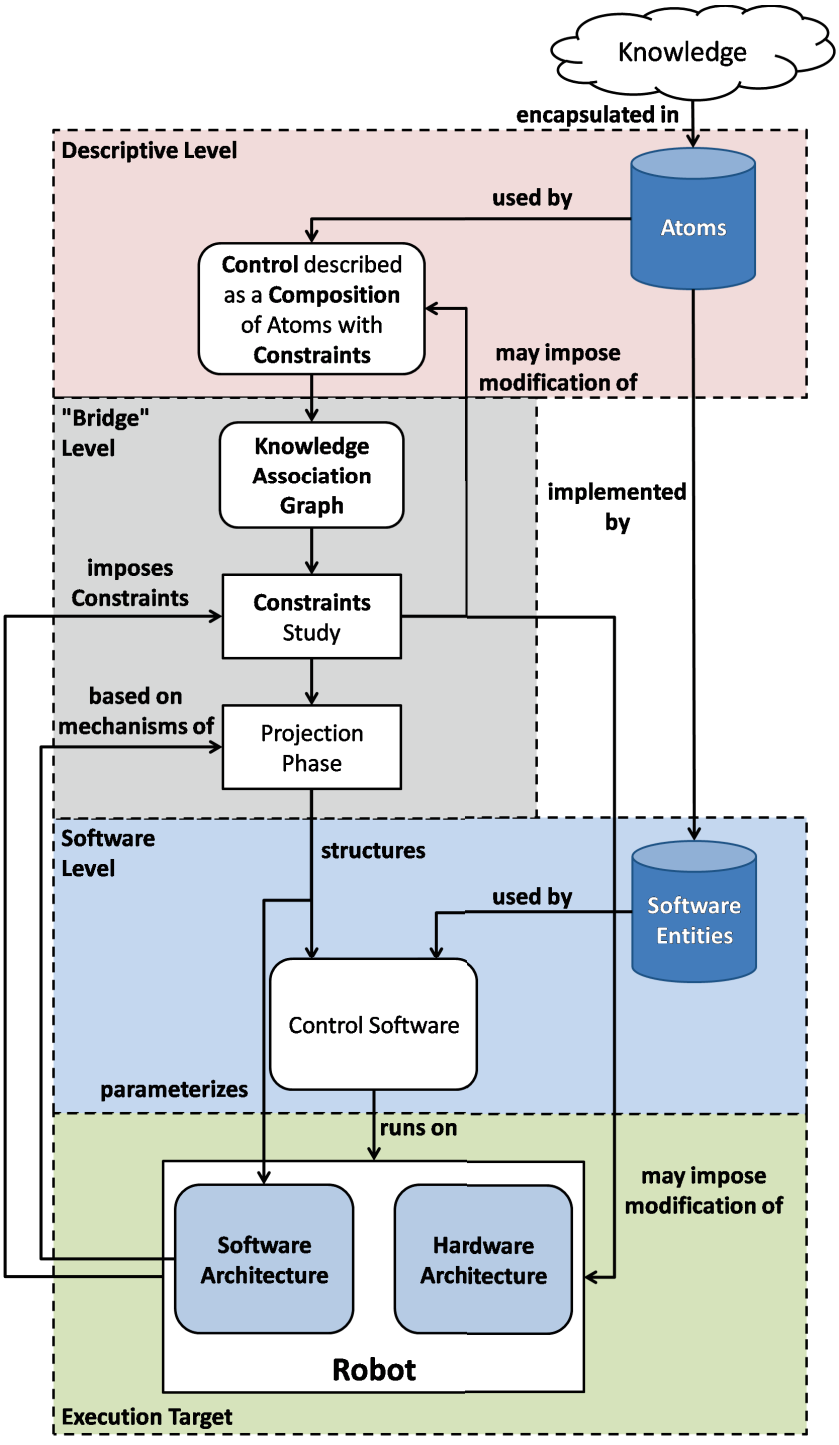

FIGURE 3.1 – Méthodologie proposée

Deuxième partie

Description modulaire d'un contrôle : concepts et entités manipulées

Chapitre 4

Connaissance et modularité, vers quelle représentation ?

Nous proposons donc un paradigme de représentation orienté connaissance. En effet, la notion de connaissance est un élément clé pour permettre, dans notre contexte applicatif, de structurer à la fois toute description basée sur un assemblage de connaissances (une loi de commande par exemple) et son implémentation informatique (le logiciel de contrôle en robotique). Néanmoins, pour que la connaissance puisse conserver cette qualité, il faut porter une attention particulière à la représentation qui en est faite.

De plus, une description fonctionnelle de la connaissance n'est pas suffisante dès lors qu'elle doit être mise en œuvre sur une cible d'exécution. Nous devons donc introduire un moyen d'exprimer les contraintes qui émanent à la fois de la description de la connaissance (dans notre contexte, il s'agira, par exemple, de la stabilité) et celles qui nous sont imposées par notre cible d'exécution (par exemple les périodes capteurs ou la puissance de calcul du contrôleur du robot).

Ainsi, dans un premier temps, nous présenterons les critères essentiels qu'il nous faudra respecter dans notre contexte applicatif pour décrire cette connaissance. Nous définirons ensuite les entités utilisées pour encapsuler la connaissance dans notre approche et développerons les concepts associés. Une illustration concrète et détaillée de ces concepts sur un exemple simple ainsi qu'une présentation de l'API logicielle utilisée pour les mettre en œuvre sont respectivement disponibles aux Annexes A et D.

4.1 La connaissance

Notre approche se structure autour de la connaissance. Or, il s'agit d'une notion très générale qui a de nombreuses acceptions suivant le domaine dans lequel elle est utilisée. Pour

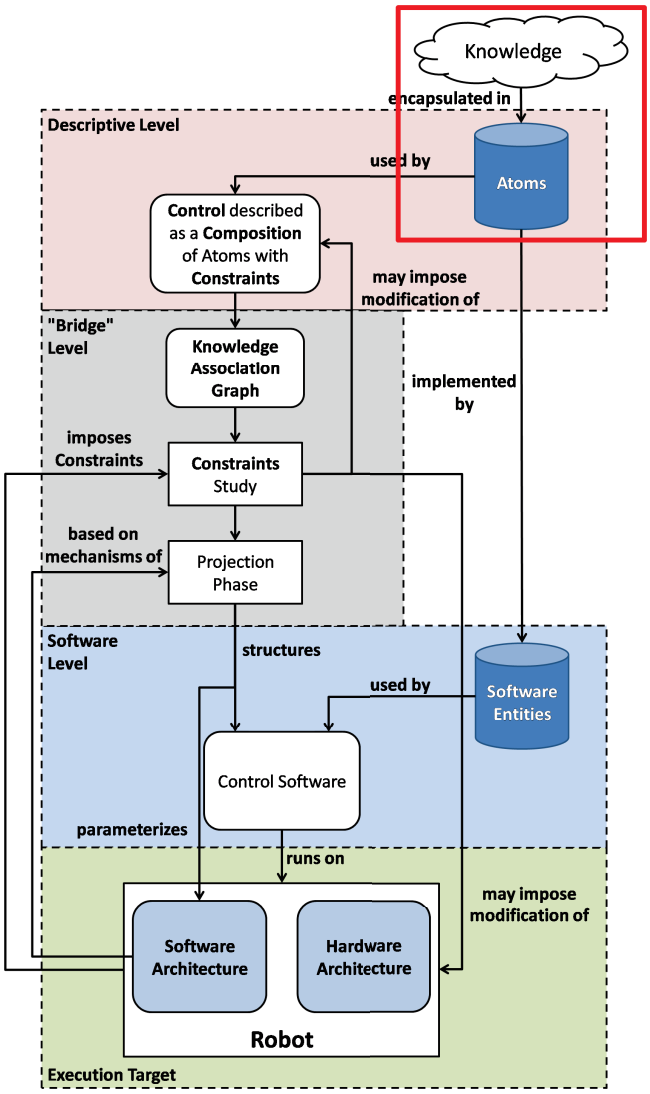

FIGURE 4.1 – Aspects de la méthodologie abordés dans le chapitre 4

définir notre représentation de la connaissance, il faut donc d'abord expliciter son sens dans notre contexte et spécifier quelles sont ses principales caractéristiques.

4.1.1 Qu'est-ce que la connaissance ?

Tout d'abord, la connaissance peut prendre des formes hétérogènes. Il peut s'agir d'un paramètre ou d'un modèle. Les modèles eux-mêmes se présentent sous des formes variées telles que des équations ou des cartes. Les représentations mathématiques associées aux modèles sont également de formes très diverses en fonction du domaine scientifique qui les a produites, des outils mathématiques dont les spécialistes disposent et des choix de représentation effectués.

De plus, la provenance des connaissances est elle aussi variée. L'automaticien développe des équations et des modèles permettant de contrôler le robot. Le fabriquant d'un robot apporte

sa connaissance sur celui-ci, par exemple ses dimensions ou sa masse. De même les fabriquants des capteurs et les spécialistes en traitement du signal apportent leurs connaissances sur ceux-ci permettant ainsi d'obtenir des informations telles que leur période d'échantillonnage, leur précision ou les traitements à appliquer aux mesures brutes pour permettre leur utilisation. Les spécialistes de l'environnement tels que des biologistes ou des hydrogéologues amènent quant à eux leurs connaissances sur l'environnement telles que des modèles a priori (comme un plan ou une topographie) mais expriment aussi leurs besoins en termes de connaissances à acquérir sur l'environnement (cartographie d'un aquifère karstique, relevés de température) et peuvent proposer des stratégies pour acquérir les connaissances dont ils ont besoin (par exemple, remonter le gradient d'une variable reconstruite).

Enfin, une connaissance n'est qu'une représentation des informations que possèdent les acteurs du développement du robot sur un phénomène ou un objet. Il ne s'agit donc bien souvent que d'une représentation partielle et plus ou moins précise d'une réalité. Cette connaissance est donc amenée à évoluer au fur et à mesure que des données sont recueillies via des expérimentations ou lorsque des avancées théoriques sont réalisées. Cela est particulièrement vrai dans le cadre de l'étude d'environnements qui mettent en jeu des dynamiques souvent complexes et dont la compréhension est souvent délicate.

De par les caractéristiques de la connaissance, il apparaît que tout assemblage de connaissances doit être réalisé de manière modulaire. Mais pour qu'une description modulaire soit réellement efficace, il faut que les entités qui encapsulent ces connaissances respectent plusieurs critères que nous devons établir en fonction de nos objectifs.

4.1.2 Un exemple d'utilisation de la connaissance en robotique : une loi de commande

Nous allons illustrer la problématique d'utilisation de la connaissance dans le cadre d'une loi de commande. A travers cet exemple nous allons chercher à décomposer une loi de commande pour comprendre comment les connaissances sont utilisées par notre commande et quelles sont les interactions entre elles.

Nous utiliserons la loi de commande présentée dans [LLA+14] pour l'évitement de parois dans un environnement de type karstique. Nous présenterons ici de manière relativement simple l'exemple qui sera développé avec plus de détails dans le chapitre 10.

Considérons un robot équipé d'un sonar profilométrique et d'un capteur de mesure de vitesse, un Loch Doppler. Notre objectif est que le robot évite les parois d'un conduit karstique afin de pouvoir naviguer en toute sécurité pour cartographier le réseau. Pour cela, notre

contrôleur va devoir déterminer les forces F_v et F_w (voir Figure 4.2) à appliquer au robot (nous n'évoquerons pas ici la transformation de ces forces en consignes à appliquer par le système d'actionnement du robot) en fonction des mesures capteurs comme le montre la Figure 4.3.

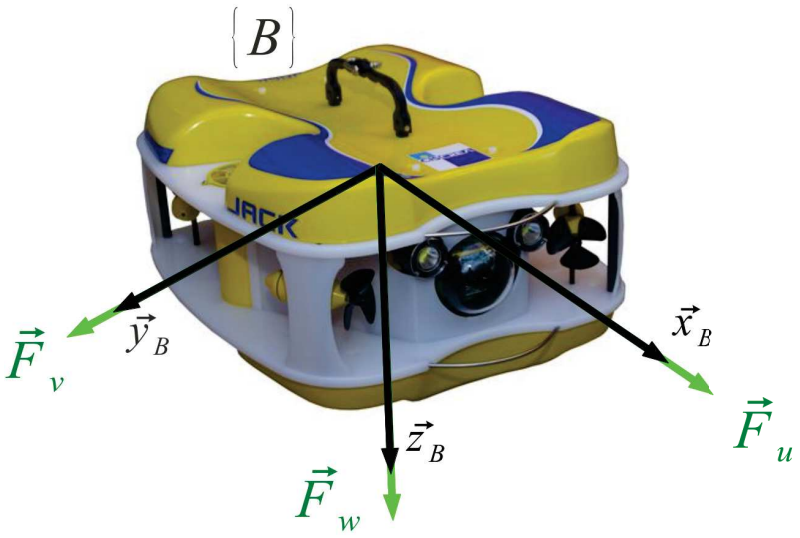

FIGURE 4.2 – Les différentes forces linéaires appliquées au robot

Le Loch Doppler nous donne les mesures de v et w (respectivement vitesses de glissement et de déplacement latéral voir Section 1.3). Le sonar profilométrique nous fournit les points d'impact avec les parois de l'environnement sous la forme d'un tableau de N_R couples $\{d_i, \alpha_i\}$, α_i étant l'angle du rayon i et d_i la distance d'impact mesurée sur ce même rayon (voir Figure 4.4), sachant que le sonar projette N_R rayons.

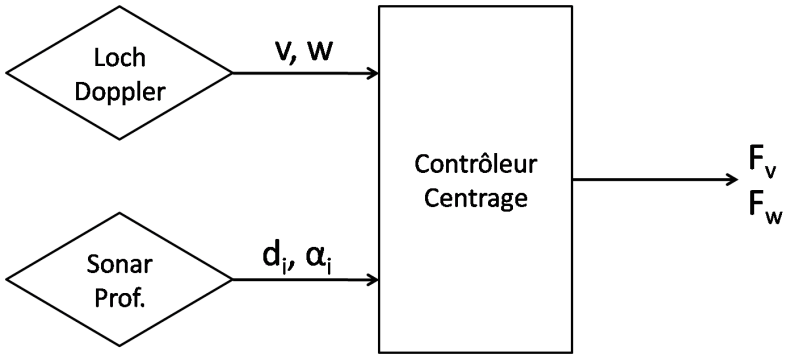

FIGURE 4.3 – Vue schématique du contrôleur monolithique

$$F_v = -K_{DVZ} \sum_{i=1}^{N_R} [cos(\alpha_i)(R_{DVZ} - d_i)] - (c_v + d_v)v \tag{4.1}$$

$$F_w = -K_{DVZ} \sum_{i=1}^{N_R} [sin(\alpha_i)(R_{DVZ} - d_i)] - (c_w + d_w)w \tag{4.2}$$

Les équations (4.1) et (4.2) illustrent bien que les différentes connaissances mises en jeu dans le contrôle sous forme monolithique sont assez peu explicites pour tout autre personne que le concepteur de la loi de commande. Décomposons cette loi de commande pour faire ressortir les différentes connaissances utilisées. Une représentation schématique de cette décomposition est proposée à la Figure 4.5.

Tout d'abord, le sonar profilométrique (Figure 4.5, Bloc A) nous fournit un ensemble de mesures de distance, chacune décrite sous la forme d'un couple (d_i, α_i) où d_i est la distance à l'obstacle et α_i est l'angle du rayon (Figure 4.4). Nous allons également utiliser deux paramètres du sonar (Figure 4.5, Bloc B) : N_R, le nombre de rayons qu'il projette et d_{max}, la portée du capteur.

Un second capteur, le Loch Doppler (Figure 4.5, Bloc C), nous fournit les mesures de vitesse du robot.

Nous définissons ensuite une zone virtuelle circulaire autour du robot inspirée des travaux sur la Zone Virtuelle Déformable (ZVD) introduits dans [ZL93]. Le rayon de cette zone, R_{DVZ}, est égal à la portée du capteur d_{max} (Figure 4.5, Bloc D et illustré Figure 4.4) car comme montré dans [LLA+14], cela permet de s'assurer que l'amplitude de la force induite par les intrusions dans la zone virtuelle sera indépendante de son rayon, R_{DVZ}. Il s'agit ici clairement de la mise en relation entre une connaissance portant sur la technologie embarquée par le robot (ici un capteur) et une autre modélisant l'interaction entre le robot et son environnement. En outre, cette relation n'est clairement pas explicite dans le cadre de la description monolithique. Nous calculons ensuite les intrusions réalisées dans cette zone (Figure 4.5, Bloc E). En effet, chaque point d'impact provenant du sonar profilométrique (Figure 4.5, Bloc A) va générer une déformation de norme $\delta_i = R_{DVZ} - d_i$ dans la zone virtuelle, comme illustré dans la Figure 4.4.

Chacune de ces déformations va générer une force de réaction, similaire à celle produite par un ressort, $\vec{f_i} = -K_{DVZ}\vec{\delta_i}$ où K_{DVZ} est un paramètre de notre zone déformable (Figure 4.5, Bloc D) permettant de faire varier sa rigidité (i.e. plus la zone sera "rigide" plus grande sera la force résultant des intrusions et plus importante sera la réaction induite du robot). Après avoir calculé les déformations, nous allons déterminer la force totale résultant de celles-ci par sommation des différentes $\vec{f_i}$ (Figure 4.5, Bloc F) :

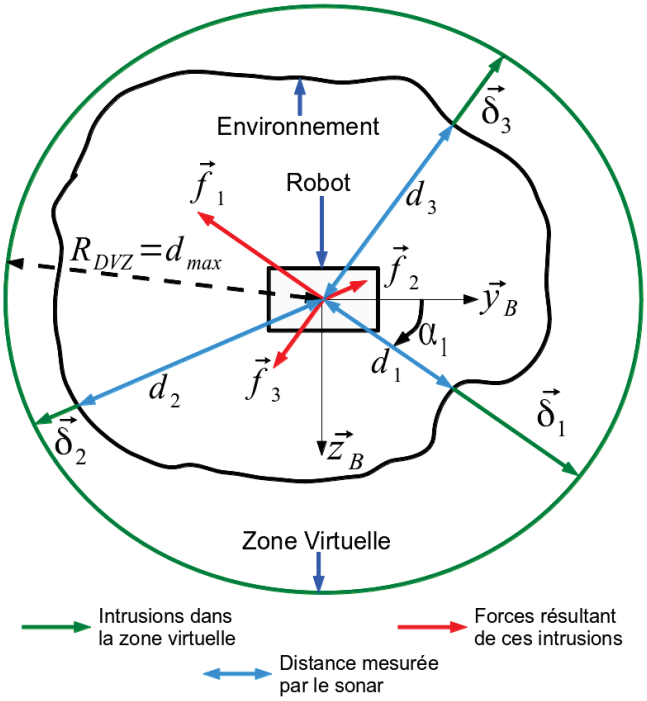

FIGURE 4.4 – Vue en coupe (plan yz) du robot dans un conduit

$$F_{R_y} = -K_{DVZ} \sum_{i=1}^{N_R} cos(\alpha_i)\delta_i \tag{4.3}$$

$$F_{R_z} = -K_{DVZ} \sum_{i=1}^{N_R} sin(\alpha_i)\delta_i \tag{4.4}$$

Nous allons donc chercher à atteindre le point d'équilibre entre ces différentes forces, c'est-à-dire le point tel que $F_{R_y} = 0$ et $F_{R_z} = 0$, évitant de fait les parois du conduit.

Pour cela, nous allons déterminer les accélérations à appliquer pour à la fois annuler les forces et atteindre ce point avec une vitesse nulle. Le robot étant iso-actionné, les déplacements suivant ses axes $\vec{y}_B$ et $\vec{z}_B$ sont découplés. De fait, annuler la force résultante calculée précédemment (Figure 4.5, Bloc F), revient à annuler chacune de ces composantes de manière indépendante. Nous utilisons une équation d'ordre 2 de type masse-ressort-amortisseur pour chaque axe (Figure 4.5, Blocs G et H) :

$$\dot{v}_{co} = \frac{F_{R_y}}{m_v} - \frac{c_v v}{m_v} \tag{4.5}$$

$$\dot{w}_{co} = \frac{F_{R_z}}{m_w} - \frac{c_w w}{m_w} \tag{4.6}$$

m_v et m_w sont les masses totales (i.e. en eau) du robot suivant les différents axes de commande. Celles-ci proviennent des paramètres du modèle dynamique du robot (Figure 4.5,

Bloc K). Les coefficients d'amortissement c_v et c_w sont calculés par les formules (Figure 4.5, Blocs I et J) :

$$c_v = 2\sqrt{m_v K_{DVZ} N_R} \tag{4.7}$$

$$c_w = 2\sqrt{m_w K_{DVZ} N_R} \tag{4.8}$$

Il s'agit là encore d'une équation constituée par assemblage de connaissances portant sur des éléments très divers :

- La relation entre le robot et son environnement au travers des paramètres du modèle dynamique (Figure 4.5, Bloc K).
- La technologie embarquée sur le robot au travers du nombre de rayons projetés par le sonar (Figure 4.5, Bloc B).
- Le paramétrage d'un modèle de l'interaction entre le robot et son environnement utilisé à des fins de commande (Figure 4.5, Bloc D).

Les liaisons entre elles sont réifiées sans ambiguïté par la décomposition (Figure 4.5, Blocs I et J).

Enfin les accélérations de commande sont transmises au modèle dynamique (Figure 4.5, Bloc L). Il est chargé de calculer la force que doivent appliquer les actionneurs (Figure 4.5, Bloc M) du robot afin d'obtenir les accélérations désirées. Pour cela, il doit connaître la masse du robot en eau ainsi que les différentes forces perturbatrices induites par la relation hydrodynamique entre le robot et son environnement. Dans le modèle très simplifié présenté Figure 4.5, Bloc L, la trainée générée par le robot lorsque celui-ci se déplace dans l'eau est la seule force prise en compte. Ce modèle dynamique dépend de divers paramètres (Figure 4.5, Bloc K) qui varient d'un robot à l'autre en fonction de caractéristiques telles que sa masse à sec ou sa forme. Celles-ci sont donc affectées par le retrait ou l'ajout de capteurs par exemple. Il est donc important de pouvoir adapter rapidement ces paramètres en fonction du robot utilisé et de son équipement.

L'ensemble des connaissances utilisées et leurs interactions sont résumées dans la Figure 4.5.

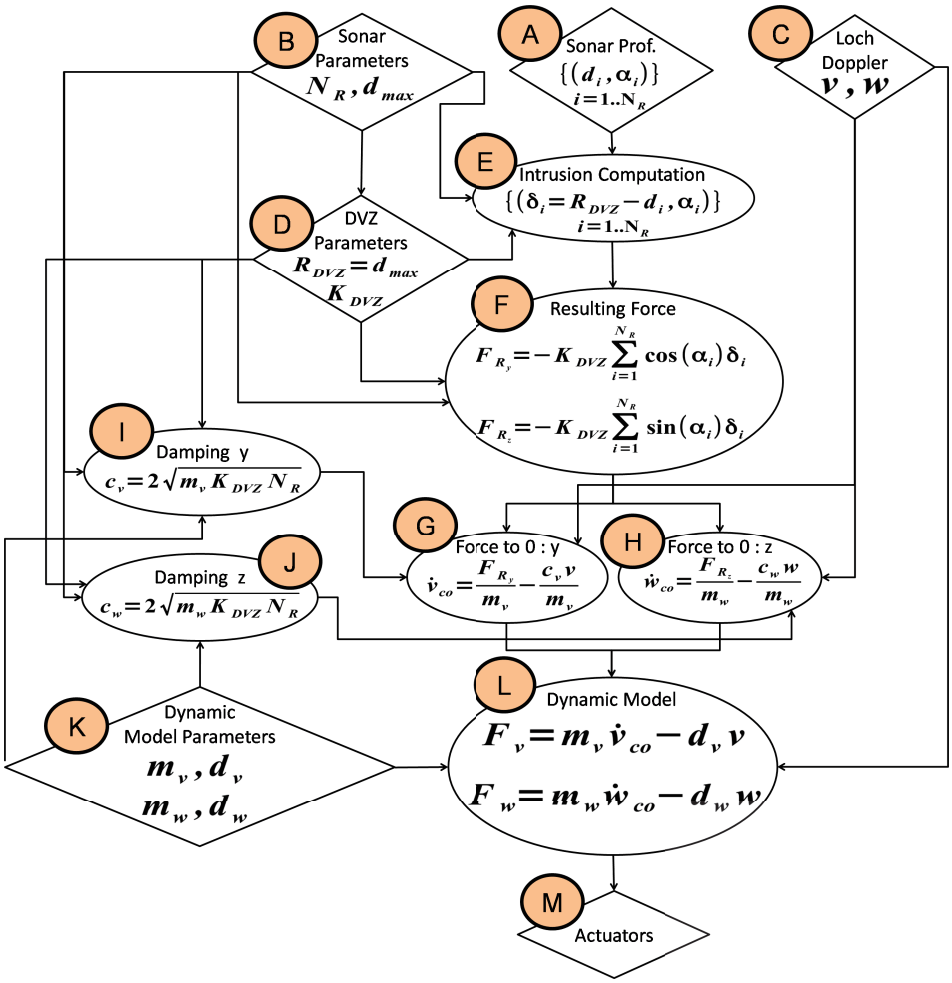

FIGURE 4.5 – Décomposition des connaissances utilisées par le contrôleur de centrage

Naturellement, l'élaboration d'une telle décomposition est un processus itératif. En effet, partant de la loi de commande monolithique, la réification de chaque connaissance et de chaque interaction est susceptible de faire apparaître au concepteur des connaissances supplémentaires qu'il est possible de décomposer. Ce processus se poursuit également bien au-delà de la phase de conception. La réutilisation de certaines connaissances dans un contexte différent (une autre architecture de contrôle par exemple) peut faire apparaître un besoin de décomposition supplémentaire de certaines connaissances nécessitant dès lors de faire évoluer l'assemblage de connaissances précédemment conçu.

4.1.3 Critères clés pour notre représentation de la connaissance

A partir de l'exemple et des points évoqués précédemment, nous pouvons identifier les caractéristiques souhaitées pour notre représentation de la connaissance.

La décomposition de l'exemple précédent illustre les différentes liaisons qui existent entre

ces connaissances. Si l'on veut pouvoir plus aisément adapter une loi de commande à une autre application robotique, il est donc indispensable de réifier ces liens. Cela plaide en faveur d'une description modulaire des associations de connaissances car la modularité nous permet de représenter celles-ci comme un assemblage d'entités clairement identifiées, reliées entre elles par des liaisons explicites.

Nous avons également vu que les connaissances se présentent sous des formes très diverses. Il est néanmoins souhaitable de pouvoir les manipuler de manière homogène. En outre, les connaissances et modèles associés sont amenés à évoluer tout en provenant de domaines divers. Ce faisant, ils se retrouvent manipulés par des utilisateurs qui ne sont pas familiers avec tous les domaines concernés. Cela nous pousse à structurer les entités encapsulant la connaissance autour de deux éléments distincts. D'un côté, un "cœur" va contenir les modèles. De l'autre, une interface permet de manipuler les connaissances de manière homogène et de les lier ensemble. Néanmoins, pour que cette encapsulation soit efficace et que les connaissances puissent être utilisées dans une composition modulaire, il faut accorder une attention toute particulière à la conception de l'interface et aux informations qu'elle exhibe. Elle doit en effet être "autosuffisante" afin qu'il ne soit pas nécessaire de connaître et de comprendre les modèles encapsulés pour pouvoir utiliser ces entités.

Enfin, dans le contexte robotique, il faut se souvenir que la description réalisée doit ensuite être implémentée sous forme de logiciel de contrôle. Nos entités doivent donc également contenir les informations pertinentes pour leur mise en œuvre.

4.2 Concepts et Définitions

Nous allons donc maintenant définir les entités issues de notre réflexion et qui permettent d'encapsuler différents éléments de connaissance.

4.2.1 Les Entités Composables

Les *Entités Composables* sont donc la base commune suivant laquelle vont être décrites nos différentes entités.

Définition 1:

Une *Entité Composable* est un 4-uplet $(Nom, Int, Phy, IntPar)$.

Nom est le nom de l'*Entité Composable*. Int est son *Interface*, Phy sa *Physique*, $IntPar$ ses *Paramètres d'Interface*.

La Figure 4.6 représente le modèle *Unified Modeling Language* (UML) d'une *Entité Composable*.

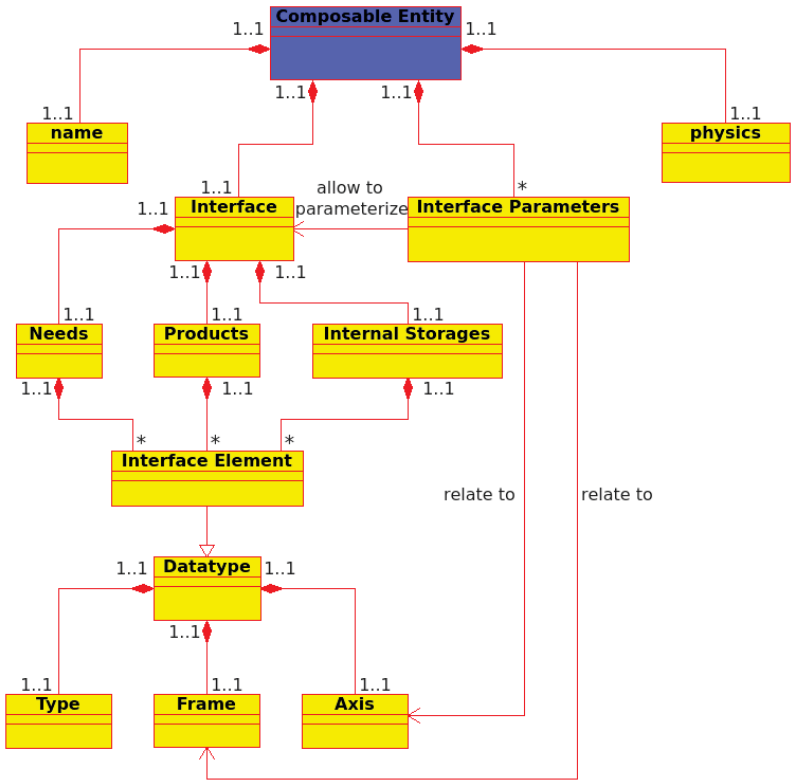

FIGURE 4.6 – Modèle UML d'une *Entité Composable*

4.2.2 Interface et Paramètres d'Interface

Définition 2:

L'*Interface* d'une *Entité Composable* est ce qui lui permet de se lier à d'autres *Entités Composables*. Il s'agit d'un 3-uplet : $Int = (Ne, Pr, IS)$. On notera l'*Interface* d'une *Entité Composable* e : $Int(e)$.

Définition 3:

Ne est l'ensemble des *Besoins* (Needs). Il s'agit des connaissances requises par l'*Entité Composable*. On notera les *Besoins* d'une *Entité Composable* e : $Ne(e)$.

Définition 4:

Pr est l'ensemble des *Produits* (Products). Il s'agit des connaissances générées par l'*Entité Composable*. On notera les *Produits* d'une *Entité Composable* e : $Pr(e)$.

Définition 5:

IS est l'ensemble des *Stockages Internes* (Internal Storages). Il s'agit des connaissances stockées par l'*Entité Composable* (pour réaliser une intégration ou dans le cadre d'historisation de données par exemple) et potentiellement accessibles depuis l'extérieur de l'*Entité Composable*. On notera les *Stockages Internes* d'une *Entité Composable* e : $IS(e)$.

Définition 6:

Un *Elément d'Interface*, i, est un élément appartenant à l'*Interface* d'une *Entité Composable*. i est donc un *Elément d'Interface* de l'*Entité Composable* e si et seulement si $i \in Ne(e) \cup Pr(e) \cup IS(e)$.
Un *Elément d'Interface* est décrit comme le couple $i = (Nom, D)$ où Nom est son nom et D l'ensemble des informations le caractérisant.

Définition 7:

L'ensemble des informations caractérisant un *Elément d'Interface* i est appelé un *Datatype* et sera noté $D(i)$. Il s'agit d'un 3-uplet $D = (Type, Frame, Axis)$

Gardant à l'esprit notre objectif de rendre l'*Interface* autosuffisante, il faut que le *Datatype* contienne toutes les informations pertinentes pour pouvoir comprendre ce que représentent les différents éléments de l'*Interface* et vérifier la cohérence des liaisons entre nos *Entités Composables*.

Définition 8:

Le *Type* désigne la grandeur représentée qu'il s'agisse d'une grandeur avec unité (Force, Accélération, Vitesse par exemple) ou sans unité (Booléen, Entier non signé par exemple). L'ensemble des *Types* définis sera noté Ty.
On a $Ty = Ty_{si} \cup Ty_{co} \cup Ty_{ar}$. Le *Type* d'un *Datatype* D sera noté $D.Type$.

Définition 9:

Ty_{si} est l'ensemble des *Types* simples. Ceux-ci ne contiennent qu'une seule grandeur.

Définition 10:

Ty_{co} est l'ensemble des *Types* complexes. Ceux-ci correspondent à une agrégation de plusieurs *Types* simples.

Définition 11:

Ty_{ar} est l'ensemble des *Types Array* (tableaux). Ceux-ci représentent une collection d'éléments de même *Type*. Ces *Types* sont caractérisés par une information de taille (i.e. la taille du tableau). Ainsi, $size(Ta)$ désignera la taille d'un *Type Array* Ta.

Il est important de noter que la taille des tableaux peut être dynamique suivant les besoins de l'application.

L'information de *Type* est-elle suffisante dans notre contexte pour caractériser précisément les éléments de l'*Interface* ?

Considérons l'exemple d'évitement de parois présenté section 4.1.2. Le sonar profilométrique effectue des mesures qu'il exprime dans un repère qui lui est propre. Il faut donc se demander dans quel repère est effectué le contrôle. Plusieurs hypothèses sont possibles, on peut par exemple envisager :

- Le contrôle est effectué dans l'espace sonar
- Le contrôle est effectué dans le repère robot et les mesures du sonar sont passées dans le repère robot au sein de l'entité sonar.
- Le contrôle est effectué dans le repère robot et les mesures du sonar sont passées dans le repère robot au sein de l'entité effectuant le calcul des intrusions.
- Le sonar est monté de telle manière que le repère sonar et le repère robot coïncident retirant la nécessité d'une conversion.

Sans connaissance du contenu des différentes entités, il nous est impossible de savoir laquelle de ces hypothèses est valide et donc de pouvoir correctement réutiliser la loi de commande décrite dans une autre application robotique. D'autant plus qu'ici nous n'avons considéré que la relation entre deux entités. La même problématique est soulevée pour chaque lien entre des entités. Comme souligné dans le cadre de ROS (voir Section 2.4.1 et [Foo13]), il s'agit là d'une problématique importante qui est fréquemment source d'erreurs dans le développement d'une architecture de contrôle.

Cela contrevient donc à notre volonté d'avoir une *Interface* autosuffisante pour les utilisateurs. Il nous faut donc ajouter des informations supplémentaires dans notre description du *Datatype*.

Nous avons ainsi ajouté deux champs supplémentaires permettant de caractériser un *Elément d'Interface*.

Définition 12:

Frame est un champ désignant le repère (frame) dans lequel est exprimé l'*Elément d'Interface*. L'ensemble des repères existants sera noté Fr. Le *Frame* d'un *Datatype* D sera noté $D.Frame$.

Différentes valeurs sont possibles pour ce champ. Trois valeurs sont obligatoirement définies :

- $NOFRAME$: la donnée n'appartient à aucun repère particulier.
- $WORLD$: la donnée appartient au repère monde.
- Une valeur est définie par robot. Dans notre cas, nous n'utiliserons qu'un seul engin. Nous noterons donc cette valeur $ROBOT$. La donnée est alors exprimée dans le repère robot.

Il est également nécessaire de définir des repères pour chaque capteur donnant des mesures relatives à une position ou à l'un de ses dérivés (vitesse, accélération). Par exemple, un repère sera défini pour le sonar profilométrique, le Loch Doppler ou encore la centrale inertielle.

Nous définirons également des repères pour le système d'actionnement, dans notre cas un repère par moteur. Enfin, des repères spécifiques pour chaque tâche peuvent aussi être définis par l'utilisateur.

Définition 13:

Axis est un champ désignant l'axe ou le plan dans lequel est exprimé l'*Elément d'Interface*. L'ensemble des axes existants sera noté Ax. L'*Axis* d'un *Datatype* D sera noté $D.Axis$.

Nous avons défini 7 valeurs possibles pour ce champ :

- $NOAXIS$: la donnée n'est pas exprimée dans un axe particulier.
- x : la donnée évolue suivant l'axe x.
- y : la donnée évolue suivant l'axe y.
- z : la donnée évolue suivant l'axe z.
- xy : la donnée évolue dans le plan xy.
- yz : la donnée évolue dans le plan yz.
- xz : la donnée évolue dans le plan xz.

Définition 14:

Deux *Datatypes*, D_1 et D_2, seront définis comme identiques, noté $D_1 = D_2$, si et seulement si $D_1.Type = D_2.Type$ et $D_1.Frame = D_2.Frame$ et $D_1.Axis = D_2.Axis$ et $size(D_1.Type) = size(D_2.Type)$ si $D_1.Type \in Ty_{ar}$.

Dans certains cas, il est également intéressant de pouvoir adapter l'*Interface* d'une *Entité Composable* à l'utilisation qui en est faite. Considérons l'exemple de la connaissance d'annulation de force (Figure 4.5, blocs G et H). Les équations (4.5) et (4.6) sont identiques à la différence près de l'axe sur lequel nous souhaitons annuler la force. Nous pourrions faire deux *Entités Composables*, une par axe mais ce ne serait pas une solution judicieuse notamment car elle est potentiellement source d'erreurs.

Il nous faut donc pouvoir indiquer suivant quel axe sont exprimés les différents *Besoins* et *Produits*. Nous allons utiliser pour cela des *Paramètres d'Interface*.

Définition 15:

Les *Paramètres d'Interface* représentent l'ensemble des modifications possibles à l'*Interface* d'une *Entité Composable* en fonction de l'utilisation qui en sera faite. L'ensemble des *Paramètres d'Interface* d'une *Entité Composable* e, noté $IntPar(e)$, est défini comme :
$IntPar(e) = \{ip_i,\ i \in [1\ ...\ Dim(IntPar(e))]\ /\ ip_i \in Fr \cup Ax\}$

Si les *Paramètres d'Interface* permettent d'adapter repères ou axes, nous n'autorisons pas dans notre approche l'utilisation de *Paramètres d'Interface* portant sur les *Types*. En effet, nous souhaitons toujours conserver la signification physique des différents modèles. Or une trop grande généricité au niveau des *Types* rendrait les modèles trop abstraits puisque leur véritable signification ne serait apparente qu'à l'utilisation de l'*Entité Composable*.

4.2.3 Physique

Définition 16:

La *Physique* d'une *Entité Composable* e, notée $Phy(e)$, est l'application mathématique décrivant une connaissance : $Phy(e) : D(Ne(e)_1) \times ... \times D(Ne(e)_{Dim(Ne(e))}) \times D(IS(e)_1) \times ... \times D(IS(e)_{Dim(IS(e))}) \rightarrow D(Pr(e)_1) \times ... \times D(Pr(e)_{Dim(Pr(e))}) \times D(IS(e)_1) \times ... \times D(IS(e)_{Dim(IS(e))})$

Le choix sémantique du terme *Physique* provient de notre volonté, évoquée précédemment, de conserver la signification des phénomènes réels (i.e. physiques) qui peuvent se produire. Evidemment, la *Physique* d'un *Atome* peut tout aussi bien représenter une connaissance "abstraite" (i.e. qui ne serait pas associée à un phénomène réel) telle que, par exemple, un traitement informatique.

4.2.4 Atomes

Définition 17:

Un *Atome* est une *Entité Composable* minimale et indivisible encapsulant un élément de connaissance. Etendant la définition d'une *Entité Composable*, un *Atome* est un 6-uplet $(Nom, Int, Phy, IntPar, Ctrs, KD)$. On notera At l'ensemble des *Atomes* définis.

$Ctrs$ représente les *Contraintes* appliquées sur l'*Atome* et KD le *Domaine de Connaissance* (Knowledge Domain) auquel appartient l'*Atome*.

La Figure 4.7 représente le modèle *Unified Modeling Language* (UML) d'un *Atome*.

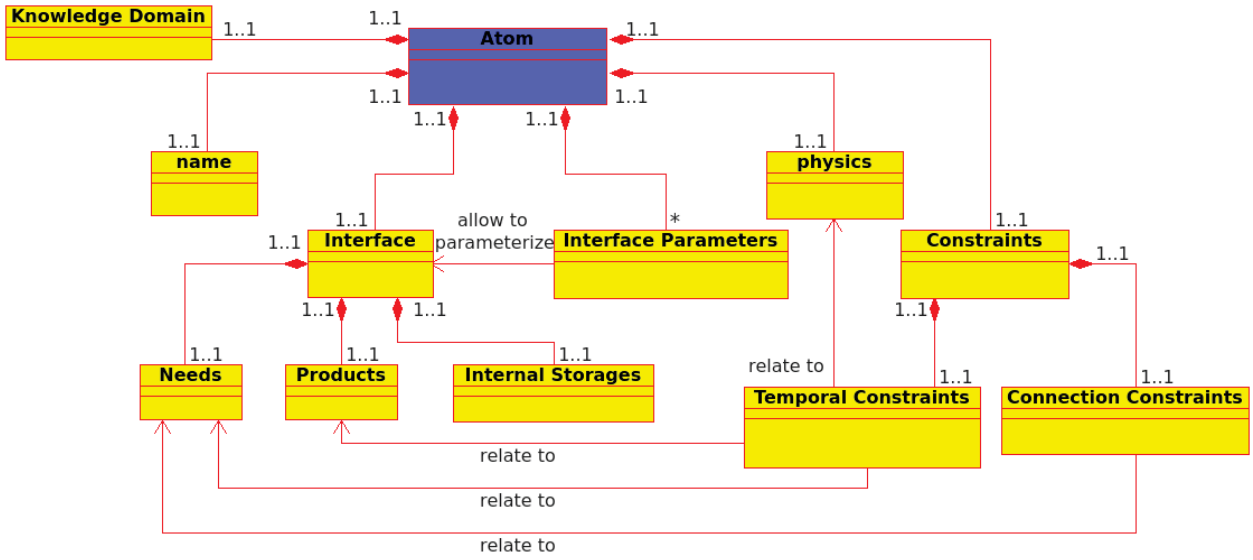

FIGURE 4.7 – Modèle UML d'un *Atome*

4.2.5 Domaines de connaissance

Définition 18:

Le *Domaine de Connaissance* d'un *Atome* a est l'information permettant d'indiquer la provenance de la connaissance représentée par l'*Atome*. Il sera noté $KD(a)$. Il s'agit d'une suite hiérarchisée de champs : $(Domain_1, \ldots, Domain_n)$ où l'ensemble des valeurs possibles pour $Domain_i$ dépend de la valeur de $Domain_{i-1}$.

Les *Domaines de Connaissance* définis dans le cadre de nos travaux sont présentés dans la Figure 4.8. Cette liste sera étendue au fur et à mesure de l'ajout de connaissances avec de nouvelles sources.

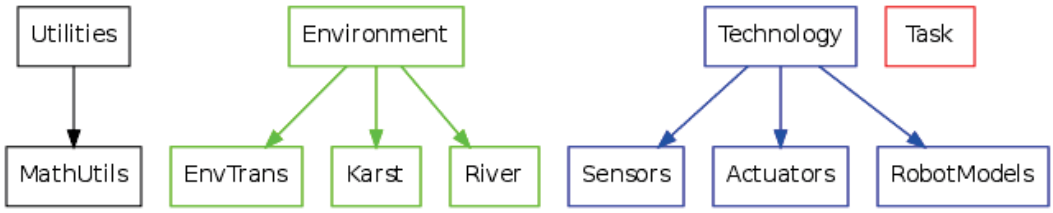

FIGURE 4.8 – *Domaines de Connaissance* définis

Les *Domaines de Connaissance* donnent des informations sur la provenance des connaissances, le domaine auquel elles se rapportent ou encore permettent de savoir qui détient l'expertise dessus. Ces informations aident à une classification des *Atomes* permettant une lecture plus facile à l'utilisateur mais ne doivent impacter en aucune façon l'utilisation faite de ceux-ci. En effet, ces informations sont sujettes à interprétation et des connaissances se retrouvent souvent à l'interface entre plusieurs *Domaines de Connaissance*.

Considérons par exemple le modèle dynamique d'un robot sous-marin. Il décrit l'interaction entre le robot et son environnement lorsque celui-ci se déplace. Il s'agit donc d'une connaissance pouvant se rapporter tout aussi bien au domaine *Technology* qu'au domaine *Environment*. Le choix de l'un ou l'autre dépend donc de l'interprétation faite par la personne ayant conçu l'*Atome*. Pour notre part, nous avons choisi de le rattacher au *Domaine de Connaissance* : *Technology : :RobotModels* car nous avons estimé que ce sont les roboticiens qui ont le plus l'habitude de l'utiliser et le plus de compréhension de ce modèle.

4.2.6 Contraintes

Définition 19:

Les *Contraintes* portant sur un *Atome* a, notées $Ctrs(a)$, sont l'ensemble des informations permettant de caractériser l'utilisation faite de l'*Atome*. De manière générale, une *Contrainte* est un couple $(nature, propriétés)$, avec l'ensemble *propriétés* dépendant de *nature*. Actuellement deux ensembles de *Contraintes* existent : *Contraintes* de connexion et *Contraintes* temporelles.

Définition 20:

La *Contrainte* de connexion portant sur un *Atome* a, notée $Ct_{co}(a)$, est un ensemble de *Contraintes* portant sur ses *Besoins* afin de déterminer s'il s'agit de *Besoins* Obligatoires (i.e. qui doivent toujours être connectés lorsque l'*Atome* est utilisé) ou Optionnels. Si la *Contrainte* est Optionnelle, on lui associe une propriété $default$ qui contiendra la valeur par défaut attribuée au *Besoin* s'il n'est pas connecté. Aucune propriété n'est associée à une *Contrainte* Obligatoire.
Soit un *Atome* a. On a $Dim(Ct_{co}(a)) = Dim(Ne(a))$. On notera $Ob(a) \subset Ne(a)$ l'ensemble des *Besoins* Obligatoires de a et $Op(a) \subset Ne(a)$ l'ensemble des *Besoins* Optionnels.
On a les relations suivantes :
$Ne(a) = Ob(a) \cup Op(a)$
$Ob(a) \cap Op(a) = \emptyset$

Définition 21:

La *Contrainte* temporelle portant sur un *Atome* a, notée $Ct_t(a)$, est un ensemble de *Contraintes* portant sur ses *Besoins*, sa *Physique* et ses *Produits*. La *Contrainte* sur la *Physique* décrit les modalités temporelles de calcul de la *Physique* et de mise à jour de ses *Produits*. Les *Contraintes* sur les *Besoins* indiquent les modalités de rafraîchissement des valeurs des *Besoins* afin de s'assurer de la validité des calculs de la *Physique*.
Soit un *Atome* a, on a $Ct_t(p) = Ct_t(Phy(a)) \ \forall \ p \in Pr(a)$. Ainsi, on ne définira jamais les *Contraintes* temporelles portant sur les *Produits* puisqu'elles sont toutes identiques à celles portant sur la *Physique*.
Le Tableau 4.1 présente les listes des *natures* et des *propriétés* des *Contraintes* temporelles portant sur les *Besoins* et le Tableau 4.2 présente celles portant sur la *Physique*.

Tableau 4.1 – *Contraintes* temporelles sur un *Besoin*

Nature	Propriété
Constant	$\emptyset$
Sporadique	$\emptyset$
Couplé	$\emptyset$
Périodique	T_{Need}

La *nature Constant* correspond à un *Besoin* dont la valeur est fixée une unique fois puis ne

change plus (initialisation). Si un *Besoin* a une *Contrainte* de *nature Sporadique*, cela signifie que l'on ne sait pas quand sa valeur sera mise à jour.

Une *nature Périodique* signifie que le *Besoin* doit être mis à jour de manière périodique. T_{Need} définit l'ensemble des périodes de rafraîchissement possibles pour le *Besoin*. Enfin, une *nature Couplé* indique que le *Besoin* doit être mis à jour avant chaque exécution de la *Physique* si celle-ci est *Périodique* (voir Tableau 4.2). Cela entraîne un couplage temporel entre cet *Atome* et celui qui value le *Besoin*.

Tableau 4.2 – *Contraintes* temporelles sur la *Physique*

Nature	Propriété
Constant	$\emptyset$
Sporadique	$C_{s_{phy}}$ $t_{comp_{max}}$ $t_{exe_{max}}$
Périodique	$t_{comp_{max}}$ T_{phy}

La *nature Constant* signifie que la *Physique* est exécutée une seule et unique fois.

Une *Physique* de *nature Sporadique* ne s'exécute que si certaines conditions sont réunies. Ces conditions, qui sont elles aussi des connaissances, sont représentées par une *Entité Composable* qui est identifiée par la *propriété* $C_{s_{phy}}$. Sa structure est spécifique puisqu'elle doit être telle que $Dim(Pr(C_{s_{phy}})) = 1$ et que son seul *Produit* est un booléen qui vaut vrai si la *Physique* doit s'exécuter. Cette *Contrainte* permet ainsi de décrire des stratégies de contrôle telle que le contrôle hybride [HJT12]. En outre, la *propriété* $t_{comp_{max}}$ sert à indiquer le pire cas du temps de calcul de la *Physique*. $t_{exe_{max}}$ correspond au temps maximum autorisé pour l'exécution de la *Physique*. Cette *propriété* est notamment utilisée lorsque les *Atomes* servent à décrire des transitions entre des connaissances (voir Section 5.1.4 sur les *Alternatives*) et qu'il est dès lors nécessaire d'indiquer une borne supérieure sur le temps de transition, ce qui est possible via cette *propriété*.

Enfin, une *nature Périodique* indique que la *Physique* doit s'exécuter à intervalles réguliers. L'ensemble des périodes d'exécution possibles est défini par la *propriété* T_{phy}. Dans le cadre d'une équation de contrôle, il peut s'agir de l'ensemble de périodes pour lesquelles la stabilité du système contrôlé sera garantie. Pour un capteur, T_{phy} désignera les périodes de rafraîchissement des données qu'il fournit. La *propriété* $t_{comp_{max}}$ désigne le temps de calcul de

la *Physique* dans le pire cas.

Intéressons nous maintenant à démontrer le point suivant de la Définition 21 : nous avons posé que pour un *Atome* a, on a $Ct_t(p) = Ct_t(Phy(a)) \; \forall \; p \in Pr(a)$.

Supposons qu'il existe un *Atome* a, tel que l'on puisse diviser sa *Physique* en deux parties indépendantes $Phy_1(a)$ et $Phy_2(a)$ qui valuent respectivement deux sous-ensembles des *Produits* $Pr_1(a)$ et $Pr_2(a)$ et qui soient telles que $Ct_t(Phy_1(a)) \neq Ct_t(Phy_2(a))$. On aurait alors $Ct_t(p) = Ct_t(Phy_1(a)) \; \forall \; p \in Pr_1(a)$ et $Ct_t(p) = Ct_t(Phy_2(a)) \; \forall \; p \in Pr_2(a)$.

Un tel *Atome* contreviendrait alors à la Définition 17 qui impose que les *Atomes* soient des entités indivisibles. Un tel *Atome* devrait donc être séparé en deux *Atomes* a_1 et a_2 tels que :

$$Phy(a_1) = Phy_1(a) \; et \; Pr(a_1) = Pr_1(a)$$
$$Phy(a_2) = Phy_2(a) \; et \; Pr(a_2) = Pr_2(a)$$

La Définition 21 peut donc s'appliquer à a_1 et a_2, séparément.

Définition 22:

Soit deux *Atomes* a_1 et a_2 et n et p tels que $n \in Ne(a_1)$ et $p \in Pr(a_2)$. Les *Contraintes* temporelles s'exerçant sur n et p sont dites compatibles, noté $Ct_t(p) \equiv Ct_t(n)$, si et seulement si elles ont des *natures* compatibles. Le Tableau 4.3 présente la liste des *natures* compatibles pour les *Contraintes* temporelles.

Tableau 4.3 – *Natures* compatibles de *Contraintes* temporelles dans le cadre de la liaison entre *Atomes* présentée Figure 4.9

$Ct_t(n)$	$Ct_t(p) = Ct_t(Phy(a_2))$
Constant	Constant
Sporadique	Sporadique
Couplé	Périodique
Périodique	Périodique

Définition 23:

Soit deux *Atomes* a_1 et a_2 et n et p tels que $n \in Ne(a_1)$ et $p \in Pr(a_2)$. Les *Contraintes* s'exerçant sur n et p sont dites compatibles, noté $Ctrs(p) \equiv Ctrs(n)$, si et seulement $Ct_t(p) \equiv Ct_t(n)$.

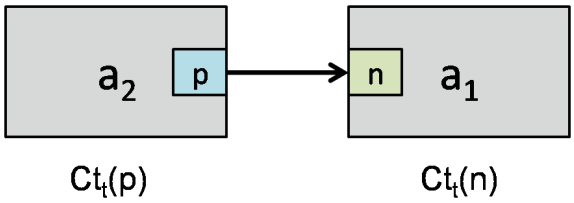

FIGURE 4.9 – Exemple de liaison entre *Atomes* et notation utilisée

Cette définition de la compatibilité de contraintes sera réutilisée plus tard dans le cadre de la description des liaisons entre *Atomes* (Section 5.1.1). En outre, elle n'intègre pas les *Contraintes* de connexion puisque celles-ci portent exclusivement sur les *Besoins*. Elles seront par contre utilisées dans le cadre de la vérification des assemblages d'*Atomes*.

4.2.7 Connaissances Externes

Lorsque l'on décrit une association de connaissances, comme c'est le cas pour une architecture de contrôle, le flux des informations n'est pas isolé du monde extérieur. Nos connaissances ont besoin d'interagir avec des entités externes à celles impliquées dans la commande décrite. Dans le cas de nos applications, il peut tout aussi bien s'agir de la valuation d'un paramètre par le concepteur de la loi de commande, de l'envoi d'une consigne via un lien de communication, d'un capteur effectuant des mesures ou de données à stocker, entre autres.

Le rôle des *Connaissances Externes* est donc de réifier ces points d'entrée ou de sortie de connaissances entre notre description et des éléments extérieurs comme illustré Figure 4.10

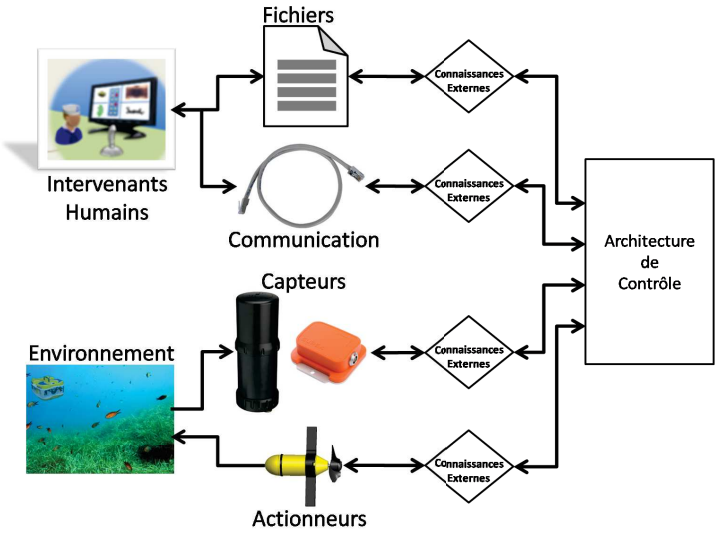

FIGURE 4.10 – Les *Connaissances Externes* réifient les points d'interaction entre un contrôleur et les entités qui lui sont extérieures

Définition 24:

On définira une *Physique* Phy comme étant indéfinie si la connaissance encapsulée dans Phy n'est pas explicitement décrite. On le notera $Phy = \emptyset$.

Définition 25:

Une *Connaissance Externe* (External Knowledge) est un *Atome* dont la *Physique* n'est pas définie et qui sert à représenter des éléments faisant le lien entre le robot et le monde extérieur. L'ensemble des *Connaissances Externes* définies est noté $EK \subset At$.
On peut ainsi écrire :
$a \in EK \iff a \in At$ et $Phy(a) = \emptyset$

Le choix de ne pas décrire la *Physique* des *Connaissances Externes* provient de notre volonté de conserver une description d'un niveau d'abstraction suffisamment élevé pour qu'elle ne dépende pas de la cible d'implémentation. En effet, de par la spécificité de leur rôle, les éléments décrits par les *Connaissances Externes* nécessitent le recours à un *driver* (capteurs ou actionneurs par exemple) ou l'utilisation d'un protocole spécifique (lien réseau, structure d'un fichier de log) qui varient d'une cible d'implémentation à l'autre.

Expliciter l'utilisation d'un *driver* ou d'un protocole dans notre *Physique* obligerait donc à définir un *Atome* pour chacune de ces entités (par exemple un *Atome* par modèle de centrale inertielle) ce qui serait difficilement maintenable sur le long terme. En outre, certains équipements nécessitent l'utilisation de différents modes (initialisation, lecture des données, mise en veille) de fonctionnement dans leur *driver* ce qui serait difficile à décrire avec la structure de nos *Atomes*.

4.3 Points clés du chapitre

- Notre approche se structure autour de la notion de connaissance.
- Celle-ci est hétérogène, amenée à évoluer dans le temps et provient de domaines d'étude très différents.
- Nous proposons un formalisme de représentation de la connaissance autour d'entités l'encapsulant pour faciliter sa manipulation et son évolution.
- Avec notre formalisme, toute association de connaissances est représentée comme une composition modulaire de ces entités.
- La structure commune à ces entités est définie par les *Entités Composables*. Elle s'organise autour d'une *Physique* qui encapsule les connaissances et d'une *Interface* qui permet de les manipuler de manière homogène.
- Les différents *Eléments d'Interface* portent toutes les informations permettant de vérifier la cohérence des liaisons entre entités facilitant donc la détection et la correction des erreurs.
- Les *Atomes* sont les entités de base. Ils portent également les *Contraintes* qui permettront de structurer notre logiciel de contrôle.

Le lecteur est également invité à se référer à l'Annexe A pour une illustration concrète et détaillée de ces différents concepts. Il peut également consulter l'Annexe D s'il souhaite découvrir comment ces concepts sont implémentés dans un langage orienté objet, le C++.

Chapitre 5

Compositions ou comment lier les Atomes ensemble

Lors du chapitre précédent, nous avons présenté les *Entités Composables* de base de notre formalisme. Il s'agit des *Atomes* qui servent à encapsuler des éléments minimaux et indivisibles de connaissance. Mais évidemment pour décrire une architecture de contrôle, il ne s'agit plus de considérer chaque connaissance de manière isolée. Nous allons donc maintenant nous intéresser à la manière de composer les connaissances entre elles afin de décrire des architectures de contrôle. Nous montrerons aussi comment notre formalisme et les informations associées à nos *Atomes* qu'il s'agisse des *Datatypes* des éléments de leurs *Interfaces* ou des *Contraintes* vont nous permettre de vérifier la validité des compositions de connaissances ainsi décrites.

De plus, comme nous travaillons à un niveau de décomposition élevé, nos assemblages contiennent de nombreux *Atomes*. Dès lors, il est également important de disposer d'*Entités Composables* permettant d'utiliser des groupes d'*Atomes* déjà associés ensemble afin de mieux structurer nos assemblages tout en permettant de réutiliser plus aisément certains *Atomes* couramment associés. En outre, un robot, au cours d'une mission, peut devoir faire face à des situations variées nécessitant des réponses spécifiques (on peut notamment citer la rencontre d'un obstacle, une panne ou encore un changement d'objectif de mission). Dès lors, un assemblage statique des connaissances n'est plus une solution utilisable. Il nous faut donc pouvoir décrire les modifications structurelles à apporter à nos assemblages de connaissances afin de les adapter à la situation rencontrée par le robot et décrire les conditions guidant le choix de la structure adéquate. Ces entités vont ainsi nous permettre d'intégrer des concepts issus du contrôle avec commutation (voir Section 2.2.1) dans nos assemblages d'*Atomes*. Nous présenterons ainsi les *Entités Composables* proposées pour répondre à ces besoins.

Une illustration concrète et détaillée de ces différents concepts est proposée dans l'Annexe B. Leur implémentation est également décrite dans l'Annexe D. Le lecteur est donc invité

à se référer à ces deux annexes pour une présentation plus approfondie de notre formalisme.

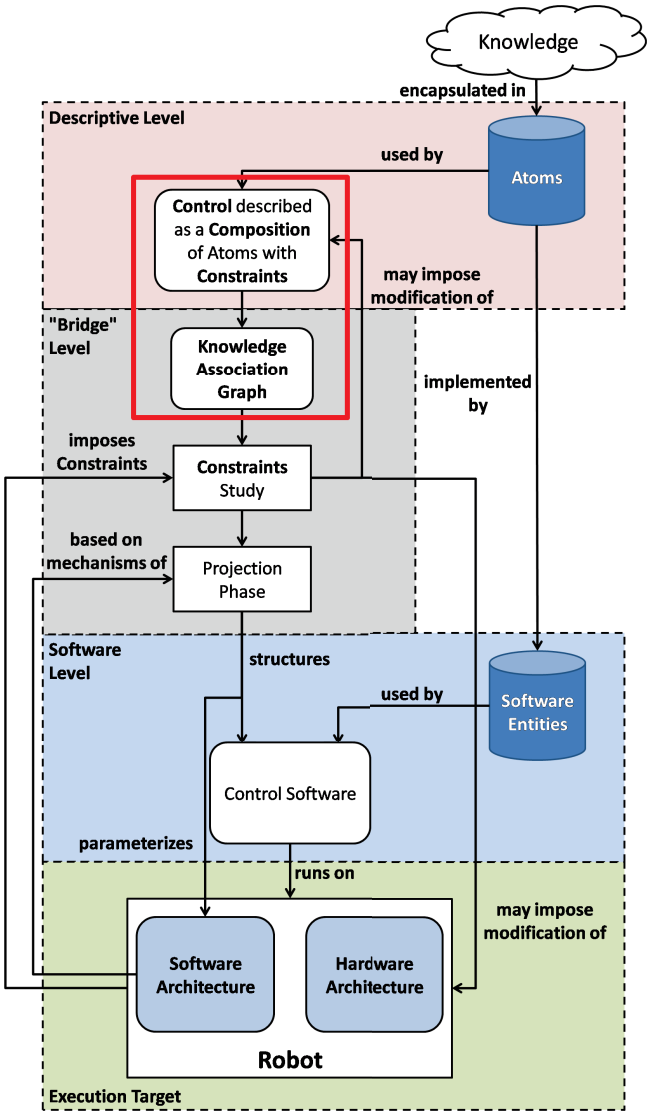

FIGURE 5.1 – Aspects de la méthodologie abordés dans le chapitre 5

5.1 Compositions entre Atomes et entités structurantes

Nous allons ici présenter les entités et concepts qui, dans notre formalisme, permettent de décrire et structurer tout assemblage de connaissances.

5.1.1 Liens

Nous avons présenté les *Atomes*, entités de base de notre approche, qui encapsulent des éléments de connaissance. Or décrire des connaissances de manière isolée est insuffisant. Il faut

pouvoir également réifier les interactions entre celles-ci. C'est le rôle des *Liens*.

Définition 26:

Soit deux *Atomes* a_1 et a_2 et n et p tels que $n \in Ne(a_1)$ et $p \in Pr(a_2)$. Un *Lien* entre n et p, noté $L(p, n)$, représente la connexion orientée de p vers n.

Définition 27:

Soit deux *Atomes* a_1 et a_2 et n et p tels que $n \in Ne(a_1)$ et $p \in Pr(a_2)$. Le *Lien* $L(p, n)$ est cohérent, noté $L(p, n) \in L_C$, avec L_C l'ensemble des *Liens* cohérents, si et seulement si $D(p) = D(n)$.

Définition 28:

Soit deux *Atomes* a_1 et a_2 et n et p tels que $n \in Ne(a_1)$ et $p \in Pr(a_2)$. Le *Lien* $L(p, n)$ est valide, noté $L(p, n) \in L_V$, avec L_V l'ensemble des *Liens* valides, si et seulement si $L(p, n) \in L_C$ et $Ctrs(p) \equiv Ctrs(n)$.

Définition 29:

Le *fan in* d'un *Besoin* n d'un *Atome* a (i.e. $n \in Ne(a)$), noté $fan\_in(n)$, représente le nombre de *Liens* connectés à ce *Besoin*.

5.1.2 Compositions

La description d'un ensemble de connaissances interagissant entre elles est appelé *Composition*. C'est cet assemblage qui va servir, dans notre contexte, à décrire des architectures de contrôle. Néanmoins avant d'associer nos *Atomes* dans une *Composition*, il est nécessaire de spécifier les modalités suivant lesquelles ils vont être utilisés.

Définition 30:

On appelle *Instanciation* le fait de valuer les *Paramètres d'Interface* et les *Contraintes* d'un *Atome*. Les entités ainsi créées sont appelées *Instances*. Chaque *Instance* est repérée par un *Identifiant*.

Définition 31:

On appelle *Composition* tout assemblage d'*Instances*. Une *Composition* c peut être définie comme la paire :
$c = (I, L)$ où I est l'ensemble des *Instances* utilisées dans la *Composition* et L l'ensemble des *Liens* entre elles.

Définition 32:

Une *Composition* est valide si et seulement si :
$fan\_in(i) = 1 \ \forall \ i \in Ob(I_j) \ \forall \ j \in [1 \ .. \ Dim(I)]$ (Tous les *Besoins* Obligatoires des *Instances* de la *Composition* doivent avoir une et une seule connexion.)
$fan\_in(i) \leq 1 \ \forall \ i \in Op(I_j) \ \forall \ j \in [1 \ .. \ Dim(I)]$ (Tous les *Besoins* Optionnels des *Instances* de la *Composition* ne peuvent pas avoir plus d'une seule connexion mais peuvent ne pas en avoir.)
$L_k \in L_V \ \forall \ k \in [1 \ .. \ Dim(L)]$ (Tous les liens de la *Composition* doivent être valides.)

5.1.3 Molécules

Afin de faciliter la description des *Compositions*, il serait intéressant de disposer d'*Entités Composables* permettant d'en regrouper d'autres. En effet, ces entités permettraient de réutiliser des ensembles d'*Entités Composables* déjà associées ce qui faciliterait la tâche du concepteur de la loi de commande. De telles entités sont appelées *Molécules*.

Définition 33:

Une *Molécule* est une *Entité Composable* encapsulant d'autres *Entités Composables*. On notera l'ensemble des *Molécules* définies Mo.
La *Physique* d'une Molécule m est définie comme le 4-uplet :
$Phy(m) = (E, \ L, \ l_n, \ l_p)$
E est l'ensemble des *Entités Composables* contenues dans la *Physique* de m et telles que $e_i \neq m \ \forall \ i \in [1 \ .. \ Dim(E)]$ (Une *Molécule* ne peut pas s'inclure elle-même.)
L est l'ensemble des *Liens* entre éléments de E.
l_n est l'ensemble des *Liens d'Interface* entre les *Besoins* de m et les *Besoins* des éléments de E.
l_p est l'ensemble des *Liens d'Interface* entre les *Produits* de m et les *Produits* des éléments de E.

On ne définit pas de *Domaines de Connaissance* pour les *Molécules*. En effet, ces dernières peuvent regrouper des connaissances provenant d'horizons très divers, et par conséquent de *Domaines de Connaissance* différents, ce qui rend alors très difficile le choix du domaine auquel les rattacher.

Définition 34:

On appelle *Lien d'Interface* le lien de référencement entre un *Besoin* d'une *Molécule* et un *Besoin* d'une entité contenue dans celle-ci ou entre un *Produit* d'une *Molécule* et un *Produit* d'une entité contenue dans celle-ci. Soit une *Molécule* m et e une entité contenue dans cette *Molécule*, le *Lien d'Interface* sera noté $l(i_m, i_e)$ avec $i_m \in Ne(m)$ et $i_e \in Ne(e)$ ou $i_m \in Pr(m)$ et $i_e \in Pr(e)$.

Définition 35:

Un *Lien d'Interface* $l(i_m, i_e)$ est cohérent, noté $l(i_m, i_e) \in l_C$, si et seulement si $D(i_m) = D(i_e)$.

Définition 36:

Une *Molécule* m, telle que $Phy(m) = (E,\ L,\ l_n,\ l_p)$, est valide, noté $m \in M_V$, si et seulement si
$fan\_in(i) = 1\ \forall\ i \in Ob(e_j)\ \forall\ j \in [1\ ..\ Dim(E)]$ (Un *Besoin* obligatoire doit recevoir une et une seule connexion.)
$fan\_in(i) \leq 1\ \forall\ i \in Op(e_j)\ \forall\ j \in [1\ ..\ Dim(E)]$ (Un *Besoin* optionnel ne doit pas recevoir plus d'une connexion mais peut ne pas être connecté.)
e_i est valide $\forall i \in [1\ ..\ Dim(E)]$
$L_j \in L_C\ \forall\ j \in [1\ ..\ Dim(L)]$ (Les *Liens* doivent être cohérents.)
$l_{n_k} \in l_C\ \forall\ k \in [1\ ..\ Dim(l_n)]$
$l_{p_o} \in l_C\ \forall\ o \in [1\ ..\ Dim(l_p)]$ (Les *Liens d'Interface* doivent être cohérents.)
$Dim(Ne(m)) \leq \sum_{i=1}^{Dim(E)} Dim(Ne(e_i))$ (La *Molécule* ne peut pas avoir plus de *Besoins* que la somme des *Besoins* des *Entités Composables* qu'elle contient.)
$Dim(Pr(m)) \leq \sum_{i=1}^{Dim(E)} Dim(Pr(e_i))$ (La *Molécule* ne peut pas avoir plus de *Produits* que la somme des *Produits* des *Entités Composables* qu'elle contient.)

La vérification de la validité d'une *Entité Composable* contenue dans une *Molécule* dépend du type d'entité dont il s'agit. Un *Atome* sera toujours valide. Une *Molécule* devra respecter les conditions ci-dessus. Nous présenterons les conditions de validité des autres *Entités Composables* au fur et à mesure de leur introduction.

Définition 37:

L'*Instanciation* d'une *Molécule* consiste à valuer ses *Paramètres d'Interface* et les *Contraintes* portant sur ses constituants.

Définition 38:

Un *Graphe Moléculaire* (Molecular Graph) est le **Graphe Orienté** décrivant la structure **interne** d'une *Molécule*. Soit m une *Molécule* telle que $Phy(m) = (E,\ L,\ l_n,\ l_p)$, on notera $MG(m) = (V, A)$ le *Graphe Moléculaire* associé tel que :
$V = E \cup Ne(m) \cup Pr(m)$ l'ensemble des noeuds (Vertices) du graphe
$A = L \cup l_n \cup l_p$ l'ensemble des arcs du graphe

5.1.4 Alternatives

Lors d'une mission, le robot peut devoir faire face à des situations variées. Pour cela, il est parfois nécessaire d'utiliser des connaissances spécifiquement adaptées à la situation rencontrée. Une description statique des connaissances utilisées et de leurs interactions, seule solution permise par les *Entités Composables* dont nous disposons pour l'instant, n'est donc plus adaptée comme illustré dans les travaux sur le contrôle avec commutations (voir section 2.2.1). Il nous faut disposer d'une entité supplémentaire permettant de spécifier les modifications structurelles à apporter à notre *Composition* ainsi que les conditions qui vont dicter ces changements. C'est le rôle des *Alternatives*.

Définition 39:

Soient n *Entités Composables* e_1 ... e_n. Elles sont dites **substituables** si et seulement si $Pr(e_1) = \ ... \ = Pr(e_n)$.

Définition 40:

Une *Alternative* est une *Entité Composable* décrivant un choix conditionné entre plusieurs *Entités Composables*. L'ensemble des *Alternatives* définies est noté *Al*. Comme toute *Entité Composable*, une *Alternative* est instanciable. Cela consiste à valuer ses *Paramètres d'Interface* et les *Contraintes* sur ses constituants. Soit une *Alternative al*, sa *Physique* est décrite comme le 7-uplet :
$Phy(al) = (E, e_{sel}, E_{jun}, L_{sel}, L_{jun}, l_n, l_p)$ avec
E est l'ensemble des *Entités Composables* contenues dans la *Physique* de *al* et telles que :
$Dim(E) \geq 2$
$e_i \neq al \ \forall \ i \in [1 \ .. \ Dim(E)]$ (Une Alternative ne peut se contenir elle-même.)
e_i et e_j sont substituables $\forall \ i, j \in [1 \ .. \ Dim(E)]$
e_{sel} l'*Entité Composable* de sélection telle que $Dim(Pr(e_{sel})) = 1$ et $Pr(e_{sel})$ soit l'identifiant de l'*Entité Composable* à utiliser.
E_{jun} l'ensemble des *Entités Composables* de jonction telles que $e_{jun_{ij}}$ décrive le passage de e_i à e_j $\forall i, j \in [1 \ .. \ Dim(E)]$ et $i \neq j$ et telles que $IS(e_j) \subset Pr(e_{jun_{ij}})$.
$Dim(E_{jun}) = \frac{Dim(E)!}{(Dim(E)-2)!}$
$Ne(al) \in Ne(e_1) \cup ... \cup Ne(e_{Dim(E)}) \cup Ne(e_{sel}) \cup Ne(e_{jun_1}) \cup ... \cup Ne(e_{jun_{Dim(E_{jun})}})$
$Pr(al) = Pr(e_1) = ... = Pr(e_{Dim(E)})$ (Les *Produits* de *al* sont ceux des *Entités Composables* contenues dans sa *Physique* et qui ont toutes les mêmes *Produits* puisque substituables.)
L_{sel} est l'ensemble des *Liens* entre le *Produit* de e_{sel} et les *Besoins* des éléments de E_{jun} qui donne l'identifiant de l'entité à utiliser.
L_{jun} représente les *Liens* entre les éléments de E et E_{jun}. Il s'agit de *Liens* particuliers. En effet, ceux-ci relient des *Produits* et des *Stockages Internes*. Ce type de *Lien* n'est possible que dans le cadre des *Alternatives*.
l_n est l'ensemble des *Liens d'Interface* entre les *Besoins* de *al* et les *Besoins* des éléments de E ainsi que de e_{sel} et E_{jun}.
l_p est l'ensemble des *Liens d'Interface* entre les *Produits* de *al* et les *Produits* des éléments de E.

La Figure 5.2 propose une vue schématique de la structure d'une *Alternative*.

Il faut également noter que notre définition n'exclue pas le fait que l'*Entité Composable* de sélection et celles de jonction soient elles-mêmes des *Alternatives*. Bien que cette configuration ne soit pas interdite et puisse même s'avérer utile dans certains cas (par exemple si elles sont

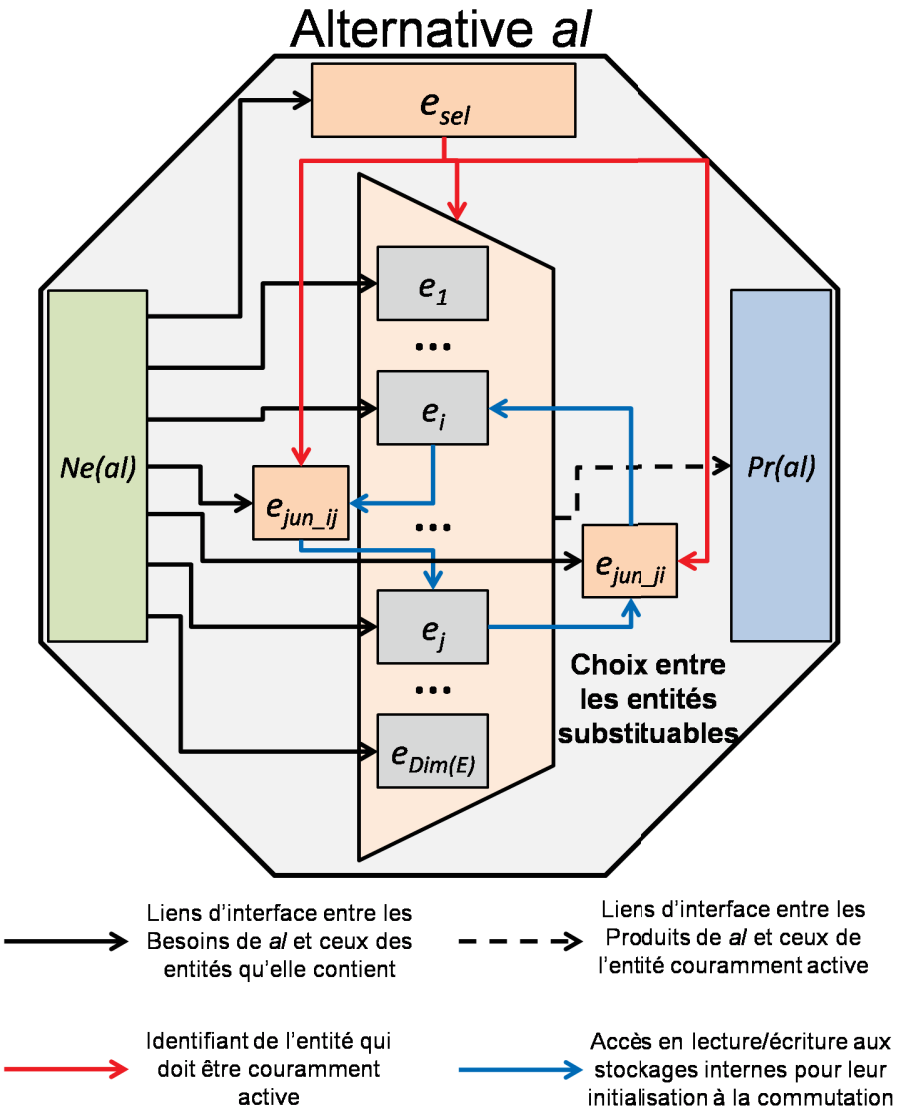

FIGURE 5.2 – Vue schématique de la structure d'une *Alternative*

basées sur plusieurs capteurs complémentaires, l'utilisation d'une *Alternative* peut servir à gérer la panne de l'un d'entre eux), la grande majorité de ces entités sera soit des *Atomes* soit des *Molécules*.

Définition 41:

Une *Alternative* al, telle que $Phy(al) = (E, e_{sel}, E_{jun}, L_{sel}, L_{jun}, l_n, l_p)$ est valide, notée $al \in AL_V$, si et seulement si
e_{sel} est valide
e_{jun_i} est valide $\forall\ i \in [1\ ..\ Dim(M_{jun})]$
e_i et e_j sont substituables $\forall\ i, j \in [1\ ..\ Dim(E)]$
e_k est valide $\forall k \in [1\ ..\ Dim(E)]$
$L_{sel_r} \in L_C\ \forall\ r \in [1\ ..\ Dim(L_{sel})]$
$L_{jun_s} \in L_C\ \forall\ s \in [1\ ..\ Dim(L_{jun})]$
$l_{n_t} \in l_C\ \forall\ t \in [1\ ..\ Dim(l_n)]$
$l_{p_u} \in l_C\ \forall\ u \in [1\ ..\ Dim(l_p)]$
$Dim(E_{jun}) = \frac{Dim(E)!}{(Dim(E)-2)!}$ (Le nombre d'*Entités Composables* de jonction est bien celui attendu.)
$Pr(al) = Pr(e_1) = ... = Pr(e_{Dim(E)})$ (Les *Produits* de al sont ceux des *Entités Composables* contenues dans sa *Physique*.)

Nous rappelons que la définition de la validité varie suivant le type d'*Entité Composable* considéré. Un *Atome* sera toujours valide. Une *Molécule* m sera valide, noté $m \in M_V$, si elle satisfait aux conditions de la Définition 36. Une *Alternative* sera valide si elle satisfait aux conditions définies ci-dessus.

5.1.5 Graphes d'Association de Connaissances

Définition 42:

Un *Graphe d'Association de Connaissances* (Knowledge Association Graph) est le **Graphe Orienté** décrivant une *Composition* telle que définie à la Définition 31. Soit c une *Composition* telle que $c = (I,\ L)$, on notera $KAG(c) = (V, A)$ le *Graphe d'Association de Connaissances* associé tel que :
$V = I$ l'ensemble des noeuds (Vertices) du graphe
$A = L$ l'ensemble des arcs du graphe

Il ne faut pas confondre la notion de *Graphe d'Association de Connaissances* et les graphes de connaissances tels que communément définis dans la littérature ([HCH15] par exemple). En effet, dans notre cas, nous ne nous préoccupons que des liaisons fonctionnelles entre nos différentes entités, c'est-à-dire que nous ne considérons que les échanges d'informations calcula-

toires entre elles, alors que les graphes de connaissances servent traditionnellement à contenir des concepts (le plus souvent abstraits) et à exprimer les liens principalement sémantiques entre eux.

5.2 Une loi de commande sous forme de Graphe d'Association de Connaissances

Pour terminer ce chapite, nous vous proposons, Figure 5.3, un exemple de *Graphe d'Association de Connaissances* qui est la traduction, selon notre formalisme, de l'exemple de la loi de commande que nous avons décrite Section 4.1.2. Pour une présentation plus détaillée de la construction de ce graphe et de ses constituants, le lecteur pourra se référer aux Annexe A et Annexe B.

Cette *Composition* nous servira de base pour illustrer notre approche dans les chapitres suivants.

5.3 Points clés du chapitre

- Une *Composition* est notre description modulaire d'un assemblage de connaissances. Les *Entités Composables* les encapsulant doivent être *Instanciées* avant d'être utilisées. Ces *Instances* sont connectées ensemble par des *Liens*.
- L'*Instanciation* consiste à spécifier la manière dont une *Entité Composable* va être utilisée dans la *Composition*.
- La définition des *Eléments d'Interface* permet de vérifier la cohérence des *Liens* et notamment des repères dans lesquels sont exprimées les connaissances. De fait, cela rend nécessaire la réification de tous les changements de repère effectués.
- Les *Molécules* sont des *Entités Composables* qui encapsulent des groupes d'entités connectées ensemble. Cela permet de faciliter leur réutilisation et mieux structurer nos *Compositions*.
- Les *Alternatives* servent à décrire des commutations entre les connaissances utilisées pour adapter nos *Compositions* à la situation rencontrée par le robot au cours de sa mission.
- Une *Composition* peut être représentée sous forme d'un graphe orienté appelé *Graphe d'Association de Connaissances*.

Le lecteur est également invité à se référer à l'Annexe B pour une illustration concrète et détaillée de ces différents concepts. Il peut également consulter l'Annexe D s'il souhaite découvrir comment ces concepts sont implémentés dans un langage orienté objet, le C++.

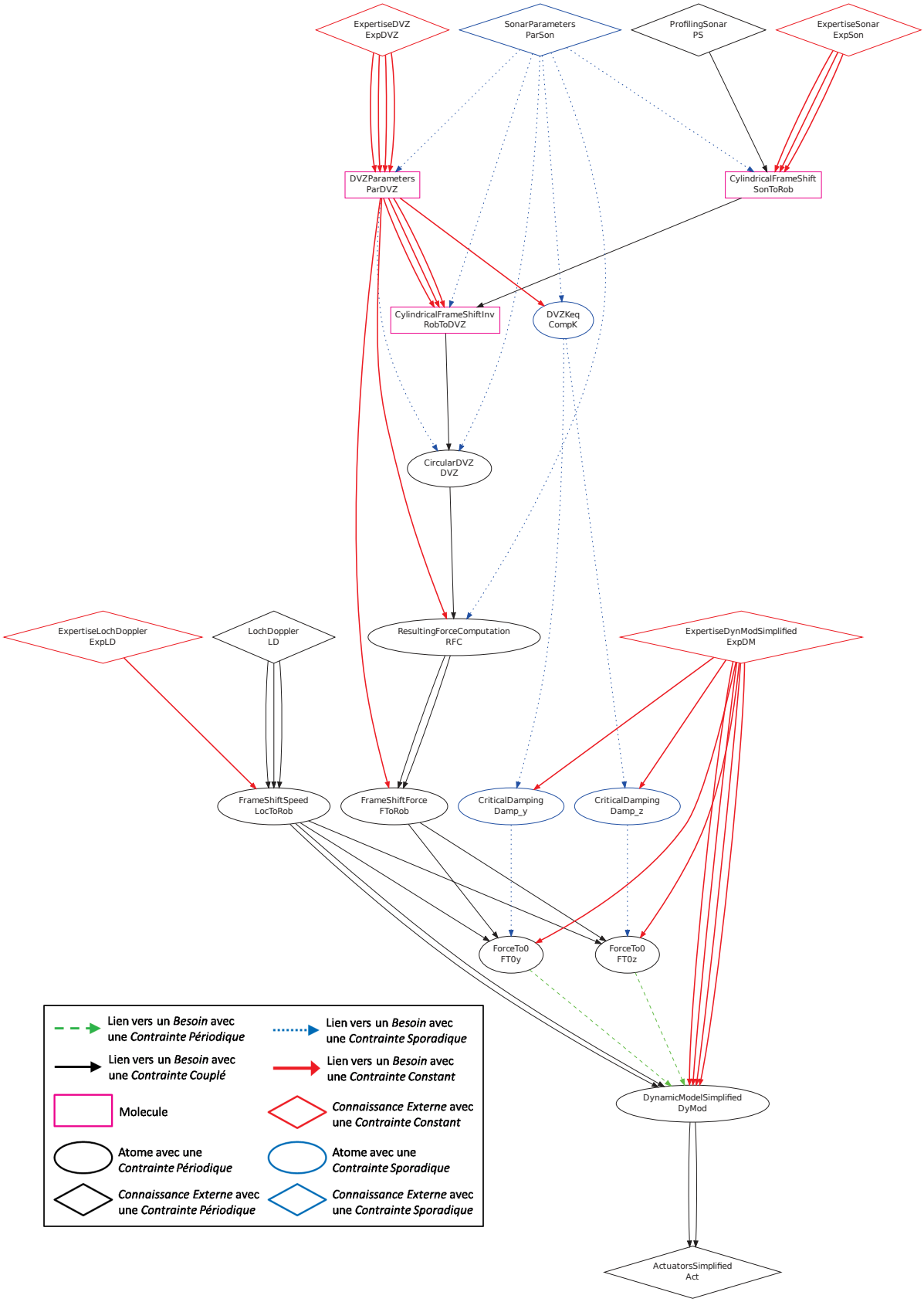

FIGURE 5.3 – *Graphe d'Association de Connaissances* représentant la fonctionnalité d'évitement de parois

Troisième partie

De la description du contrôle à son implémentation

Chapitre 6

Etude des contraintes

Dans la partie précédente, nous avons montré comment une architecture de contrôle peut être décrite comme une *Composition* d'*Entités Composables*. Cette dernière est décrite de manière indépendante de la cible d'implémentation. En outre, sur chacune des *Entités Composables*, nous faisons porter des *Contraintes* relatives à leur utilisation qui vont nous permettre de faire le lien entre la description et son implémentation.

C'est le rôle de la phase d'étude des *Contraintes* que de composer ces *Contraintes* pour qu'elles ne portent non plus sur chaque entité mais sur la *Composition* dans sa globalité. L'étude qui en résultera nous permettra d'obtenir de précieuses informations qui nous aideront ensuite à guider le concepteur dans l'implémentation et le paramétrage du logiciel de contrôle. Pour cela nous allons obtenir deux informations capitales à l'issue de cette phase. La première nous indique si le contrôle pourra être mis en œuvre sur la cible d'implémentation en respectant ou non les contraintes temporelles. Ensuite, pour les *Entités Composables* portant des *Contraintes* périodiques, cette étape nous permettra de déterminer les ensembles de périodes d'exécution admissibles en fonction des besoins des automaticiens en relation, entre autres, avec la stabilité et les capacités de la cible technologique. Il nous faudra ensuite déterminer la période d'exécution choisie au sein de l'ensemble des possibles. Enfin, ces informations vont nous permettre de guider la phase de projection qui suit directement celle-ci (voir chapitre 7) sur la base des *Contraintes* pesant sur les différentes entités.

La phase d'étude présentée ici sera orientée vers les *Contraintes* temporelles. Sa généralisation au cas d'autres *Contraintes* ne sera pas abordée dans le cadre de nos travaux.

6.1 Vue générale de la phase d'étude des Contraintes

La phase d'étude des *Contraintes* a deux objectifs majeurs. Le premier est de vérifier que le contrôle est implémentable sur notre cible technologique en respectant les contraintes de

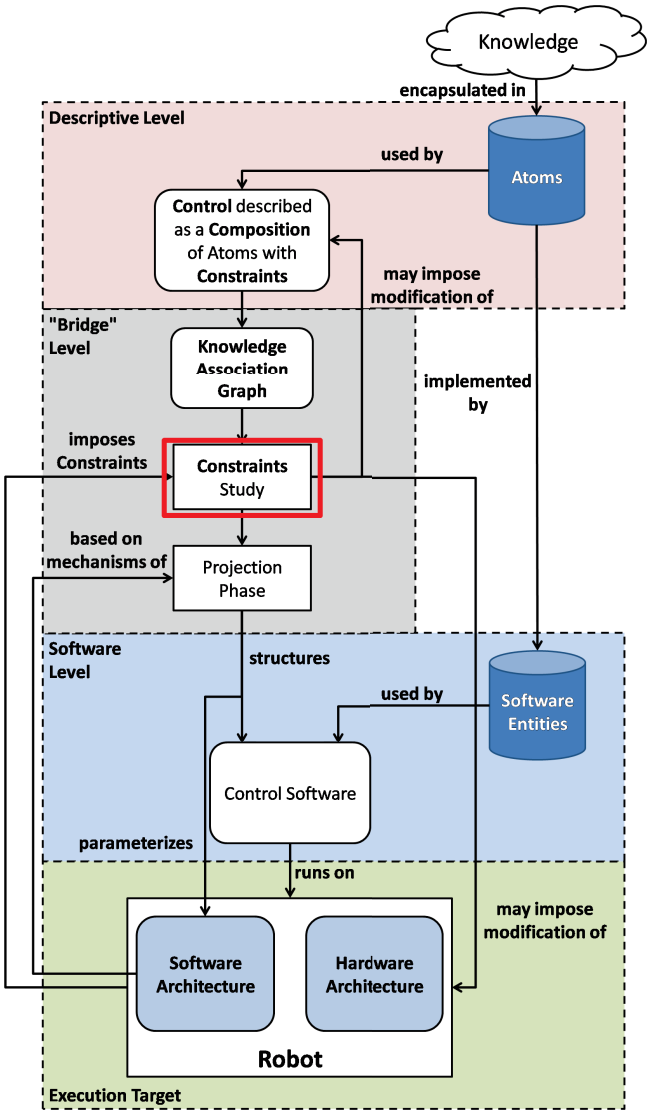

FIGURE 6.1 – Aspects de la méthodologie abordés dans le chapitre 6

stabilité fixée par l'automatique. Le second est de déterminer les ensembles de périodes d'exécution des *Entités Composables* périodiques qui nous permettent de garantir cette stabilité afin de les utiliser pour paramétrer notre logiciel de contrôle. Cette phase se déroule pour cela en différentes étapes qui sont présentées Figure 6.2.

Nous partons de la description de notre architecture de commande comme un *Graphe d'Association de Connaissances*. La première étape va consister à structurer ce graphe afin de pouvoir étudier les *Contraintes* portant sur les entités périodiques qu'il contient. Ensuite, nous allons diffuser les *Contraintes* portant sur chacune de nos entités à leur *Composition*, ce qui va nous donner les équations nous permettant de déterminer les ensembles de périodes d'exécution possibles pour nos entités.

Ces premières étapes ne sont dépendantes que de notre *Graphe d'Association de Connais-*

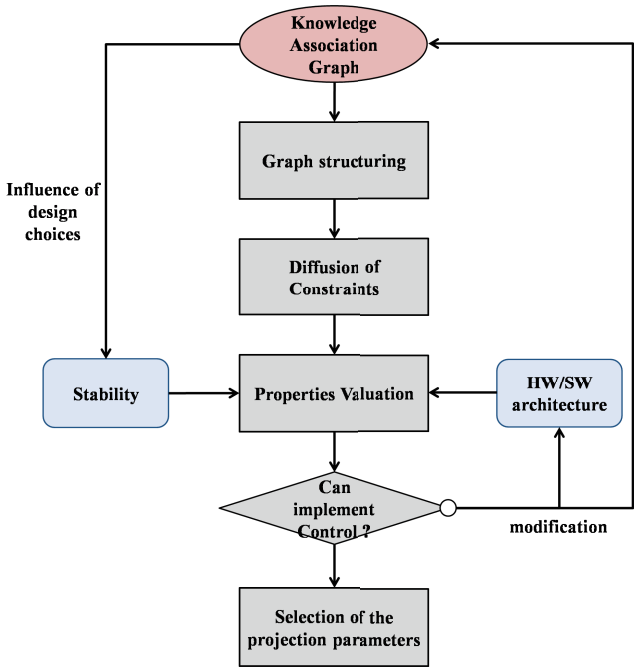

FIGURE 6.2 – Les différentes étapes à réaliser lors de l'étude des *Contraintes*

sances, elles sont donc génériques quelles que soient les caractéristiques de la cible d'implémentation.

Il nous faut ensuite valuer les *propriétés* associées à nos différentes *Contraintes*. A partir de cette étape, les opérations à réaliser sont spécifiques à la cible d'implémentation choisie et ces valeurs proviennent à la fois de la cible technologique (architecture matérielle et logicielle) et des contraintes de stabilité qui sont influencées par les choix de conception de notre *Composition*. Nous pouvons dès lors vérifier que le contrôle soit bien implémentable sur notre cible technologique. Si oui, nous pouvons alors choisir les paramètres (notamment les périodes d'exécution des différentes entités) de notre implémentation. Sinon, l'étude des *Contraintes* va nous fournir de précieuses informations sur les modifications à apporter soit à notre architecture matérielle et logicielle soit à la conception de notre *Composition*.

Nous allons maintenant détailler ces différentes étapes.

6.2 Structuration du Graphe d'Association de Connaissances

La première étape de notre méthodologie consiste à vérifier la validité de notre *Graphe d'Association de Connaissances* et, surtout, à le structurer comme présenté Figure 6.3. Tester la validité de notre *Graphe d'Association de Connaissances* revient à vérifier que la *Compo-*

sition qu'il représente est valide, au sens entendu Définition 32. Il nous faut donc vérifier les bonnes connexions de nos *Instances* (i.e. que leurs *Besoins* Obligatoires soient satisfaits et que tous, y compris ceux Optionnels, ne portent pas plus d'une connexion) ainsi que la validité des *Liens*, ce qui implique de tester leur cohérence et la compatibilité des *Contraintes*.

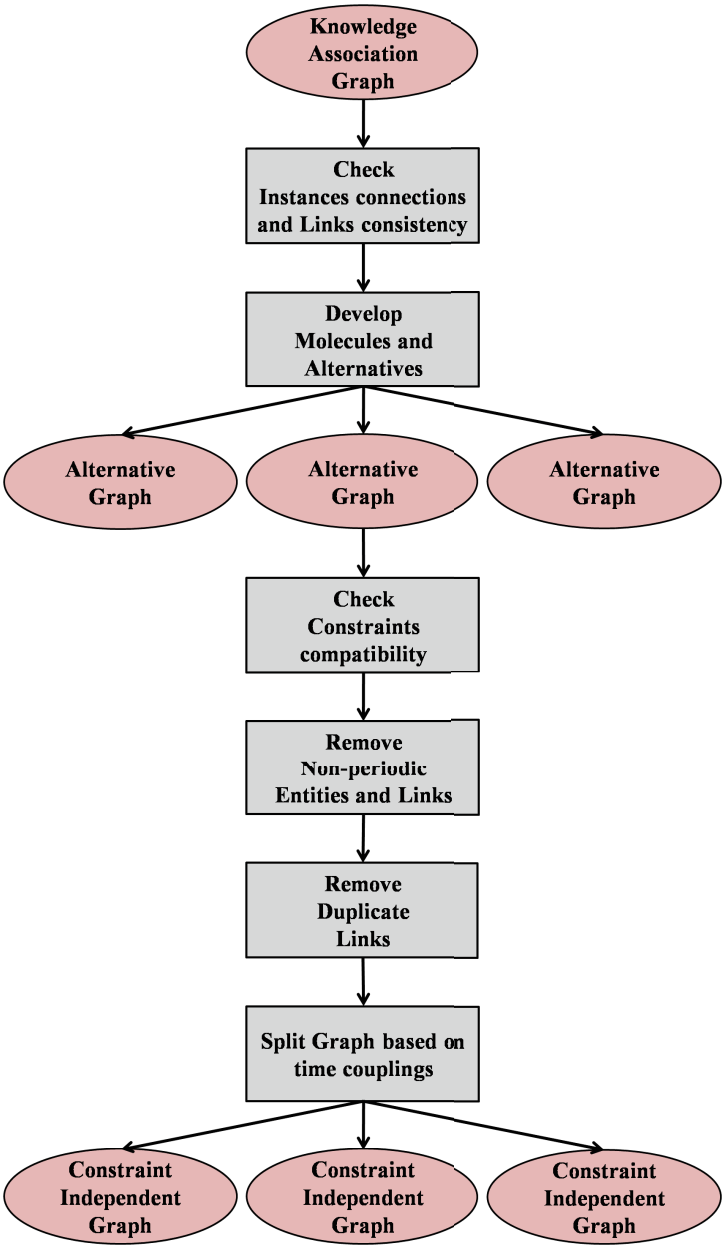

FIGURE 6.3 – Etapes de structuration du *Graphe d'Association de Connaissances*

Dans un premier temps, nous pouvons vérifier les bonnes connexions des *Instances* ainsi que la cohérence des *Liens* qui les relient (il n'est pas besoin de vérifier que les entités instanciées soient elles-mêmes valides puisque cela doit être réalisé à la déclaration des *Entités Composables*).

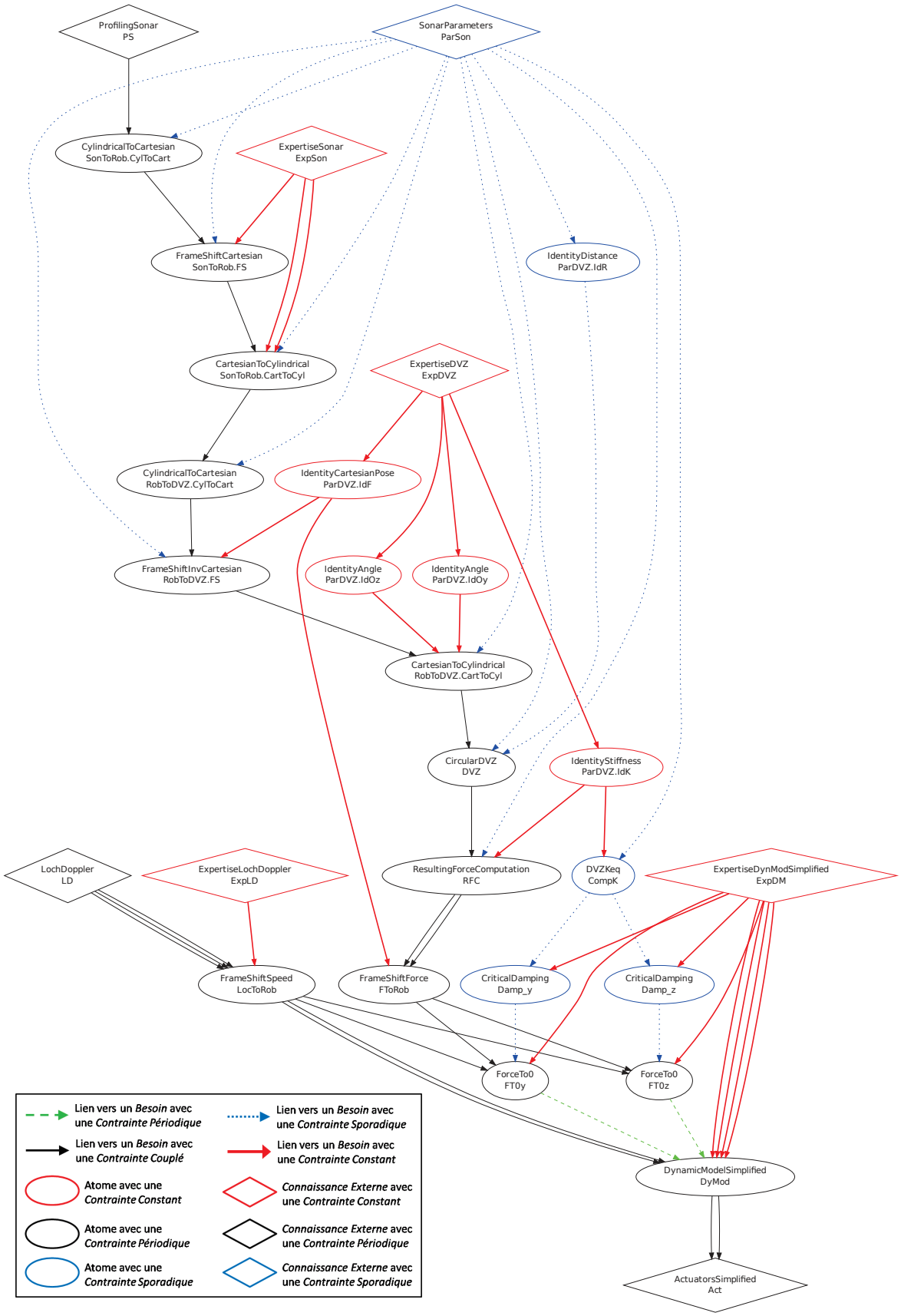

FIGURE 6.4 – Exemple de développement d'une *Molécule*

Par contre, comme expliqué à la Définition 37 et à la Définition 40 et comme souligné dans l'Annexe B, Exemple B.6, les *Contraintes* ne portent que sur les *Atomes*. De fait, avant de vérifier leur compatibilité et de poursuivre l'étude des *Contraintes*, nous allons transformer notre *Graphe d'Association de Connaissances* pour que celui-ci ne contienne plus que des *Instances* d'*Atomes*.

6.2.1 Vers un graphe fait d'Atomes

Obtenir un tel graphe nécessite de remplacer *Molécules* et *Alternatives* par leurs constituants. Cette opération est réalisée de manière itérative jusqu'à ce que le *Graphe d'Association de Connaissances* ne contienne plus que des *Instances* d'*Atomes*. Les manières de développer ces deux types d'*Entités Composables* sont différentes. Illustrons les sur des exemples.

Exemple 6.1:

Pour illustrer la manière de développer les *Molécules*, considérons la *Composition* proposée Section 5.2 (Figure 5.3) et dont les constituants sont présentés en détail dans les Annexe A et Annexe B. Le développement d'une *Molécule* implique de remplacer celle-ci par les *Entités Composables* qu'elle contient à l'aide de son *Graphe Moléculaire*. La *Composition* ainsi obtenue est représentée Figure 6.4.

Nous voyons ainsi qu'une fois les *Molécules* développées, notre *Composition* n'est plus constituée que d'*Atomes* sur lesquels les *Contraintes* sont explicites permettant ainsi leur vérification.

Exemple 6.2:

Pour illustrer la manière de développer les *Alternatives*, considérons l'*Alternative* présentée à l'Exemple B.8 (voir Annexe B) et une *Composition* basique (et partielle) l'utilisant dont les différents composants seront présentés plus loin (Chapitre 9). Cette *Composition* est représentée Figure 6.5.

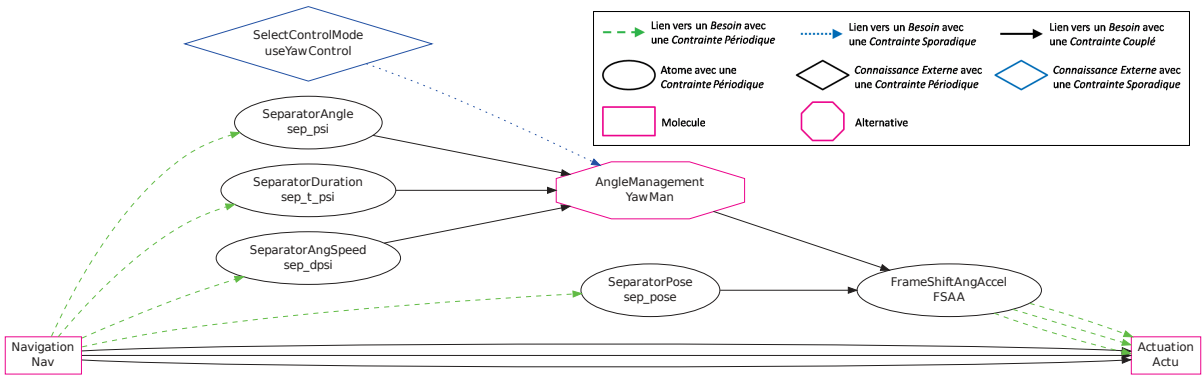

FIGURE 6.5 – Exemple de *Composition* contenant une *Alternative*

Pour développer l'*Alternative*, nous allons développer distinctement un *Graphe d'Association de Connaissances* par entité substituable de l'*Alternative*. Dans notre cas, nous allons obtenir deux graphes indépendants. Dans chacun de ces *Graphe d'Association de Connaissances*, l'*Alternative* sera remplacée par une des entités substituables et par l'*Entité Composable* de sélection.

Ainsi, dans la Figure 6.6 partie a), la *Connaissance Externe TeleopAngularAccel* et l'*Atome sélecteur* remplacent l'*Alternative* tandis que dans la partie b), c'est la *Molécule AngleControl* qui est connectée au reste de la *Composition*.

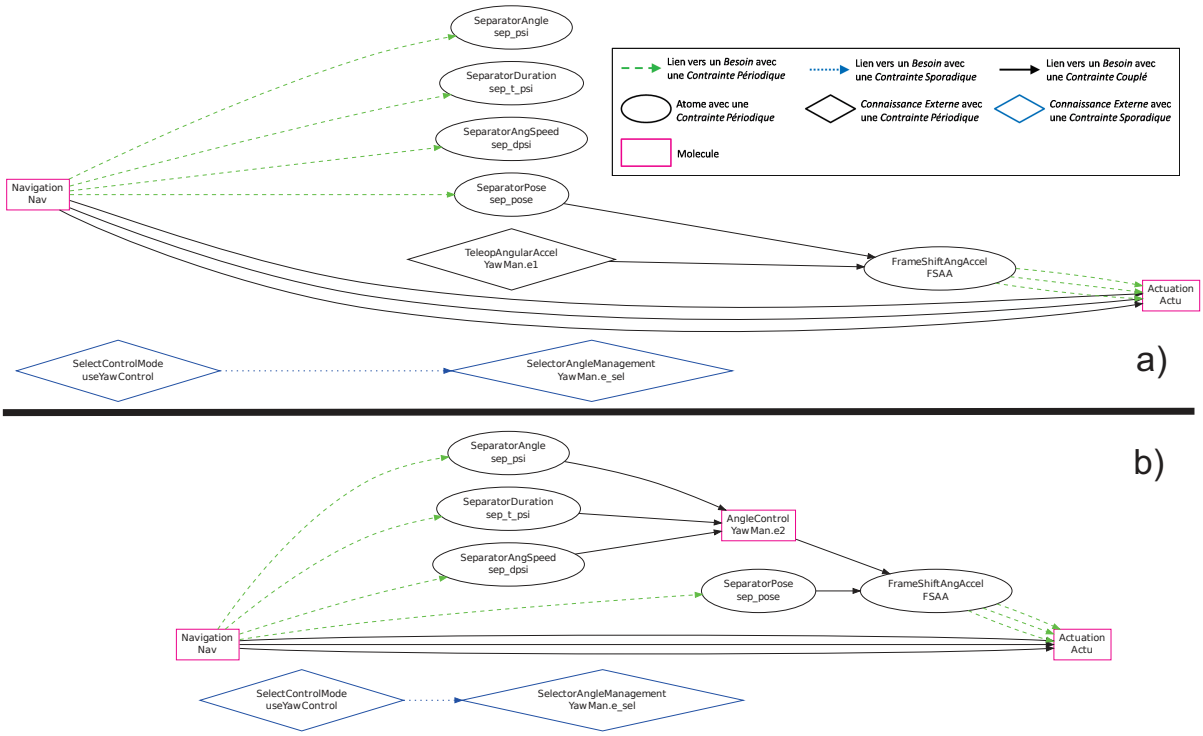

FIGURE 6.6 – Développement de la *Composition* contenant une *Alternative*

Enfin, sur les *Graphes d'Association de Connaissances* de la Figure 6.6, nous devons développer chaque *Molécule* pour obtenir le graphe constitué d'*Instances* d'*Atomes* (non représenté ici).

Ainsi, à la fin de cette partie de la restructuration du *Graphe d'Association de Connaissances*, nous obtenons un certain nombre de *Graphes d'Association de Connaissances* alternatifs entièrement constitués d'*Atomes*. Toutes les parties suivantes de la méthodologie sont à appliquer de manière indépendante sur chacun de ces *Graphes d'Association de Connaissances*.

Une fois la *Composition* décrite sous forme d'un assemblage d'*Instances* d'*Atomes*, nous pouvons finaliser sa validation en vérifiant la compatibilité des différentes *Contraintes*, comme cela a déjà été montré aux Exemples B.1 et B.6 de l'Annexe B. Une fois celle-ci validée, nous pouvons continuer les différentes étapes de structuration du *Graphe d'Association de Connaissances*.

6.2.2 Allègement de la Composition

Notre principal objectif lors de l'étude des *Contraintes* est de déterminer les périodes d'exécution des différentes entités ayant des *Contraintes* de *nature Périodique*. Nous n'avons donc besoin, dans notre *Composition*, que de ces entités là. La première étape de l'allègement consiste donc à supprimer tous les *Atomes* n'ayant pas une *Physique Périodique* ainsi que les *Liens* connectés aux *Produits* ou aux *Besoins* de ces mêmes *Atomes*.

Sur la *Composition* de la Figure 6.4, le résultat obtenu est décrit Figure 6.7.

La seconde partie de la simplification consiste à fusionner tous les *Liens* qui relient les mêmes entités en un seul. Sur la *Composition* exemple précédente, il y a deux *Liens* entre l'*Instance RFC* et l'*Instance FToRob*, entre les *Instances LocToRob* et $DyMod$, entre les $InstancesDyMod$ et *Act* et enfin trois *Liens* entre les *Instances LD* et *LochToRob*. Ceux-ci peuvent donc être regroupés, lien par lien, comme illustré Figure 6.8.

6.2.3 Séparation des Entités sur la base des couplages temporels

Une fois ceci réalisé, il est désormais nécessaire de séparer les différentes parties du graphe qui sont indépendantes temporellement. En effet, les *Contraintes* portant sur ces différentes parties pourront être étudiées indépendamment et les périodes déterminées pour chacun de ces sous-graphes.

A l'issue de cette étape, nous obtenons donc un certain nombre de graphes sur lesquels les étapes suivantes pourront être appliquées indépendamment.

Dans l'exemple de la Figure 6.8, toutes les *Instances* étant couplées temporellement, notre *Graphe d'Association de Connaissances* est le seul graphe sur lequel faire porter l'étude de ces *Contraintes*.

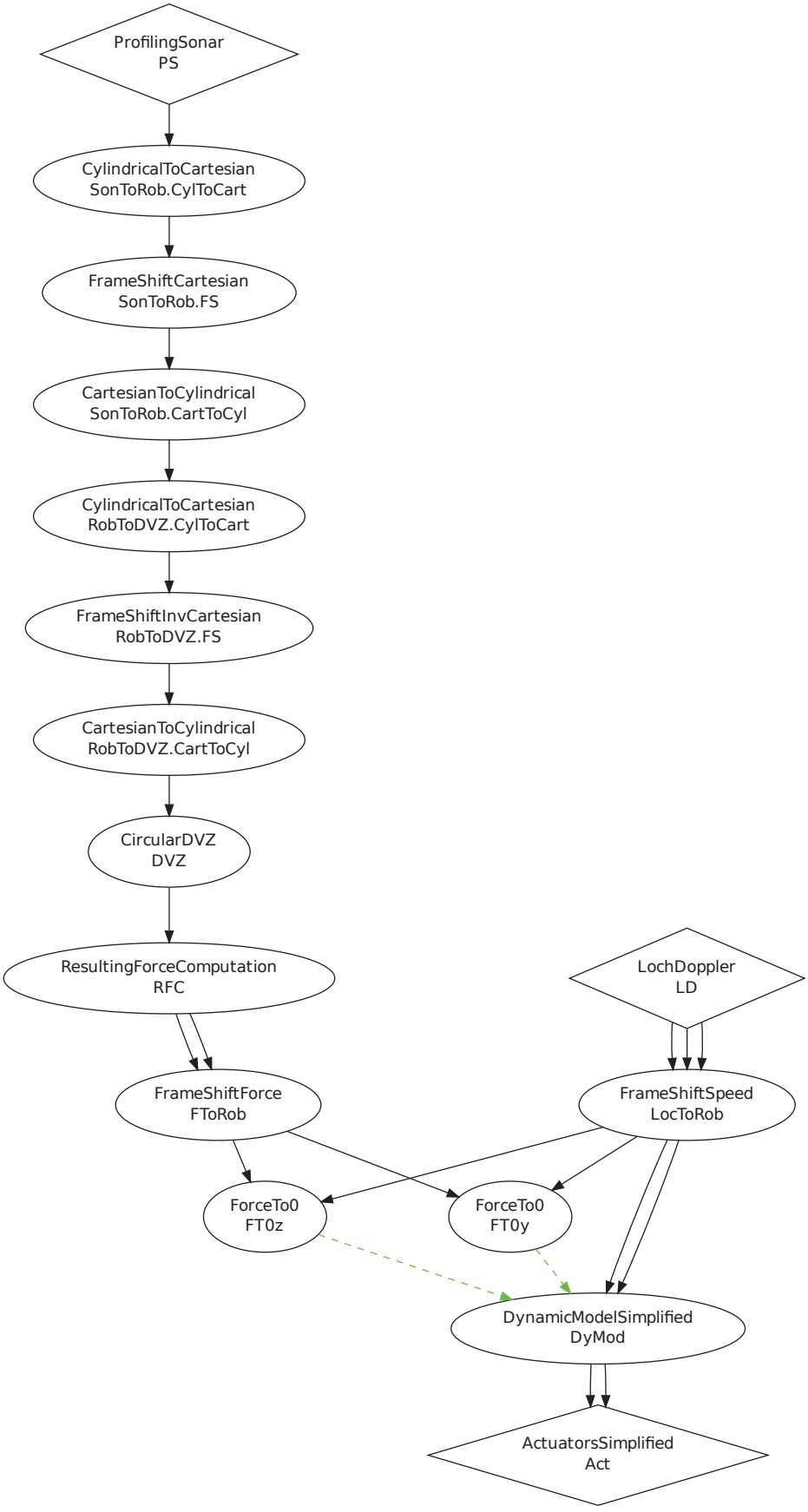

FIGURE 6.7 – Entités Périodiques du *Graphe d'Association de Connaissances* exemple

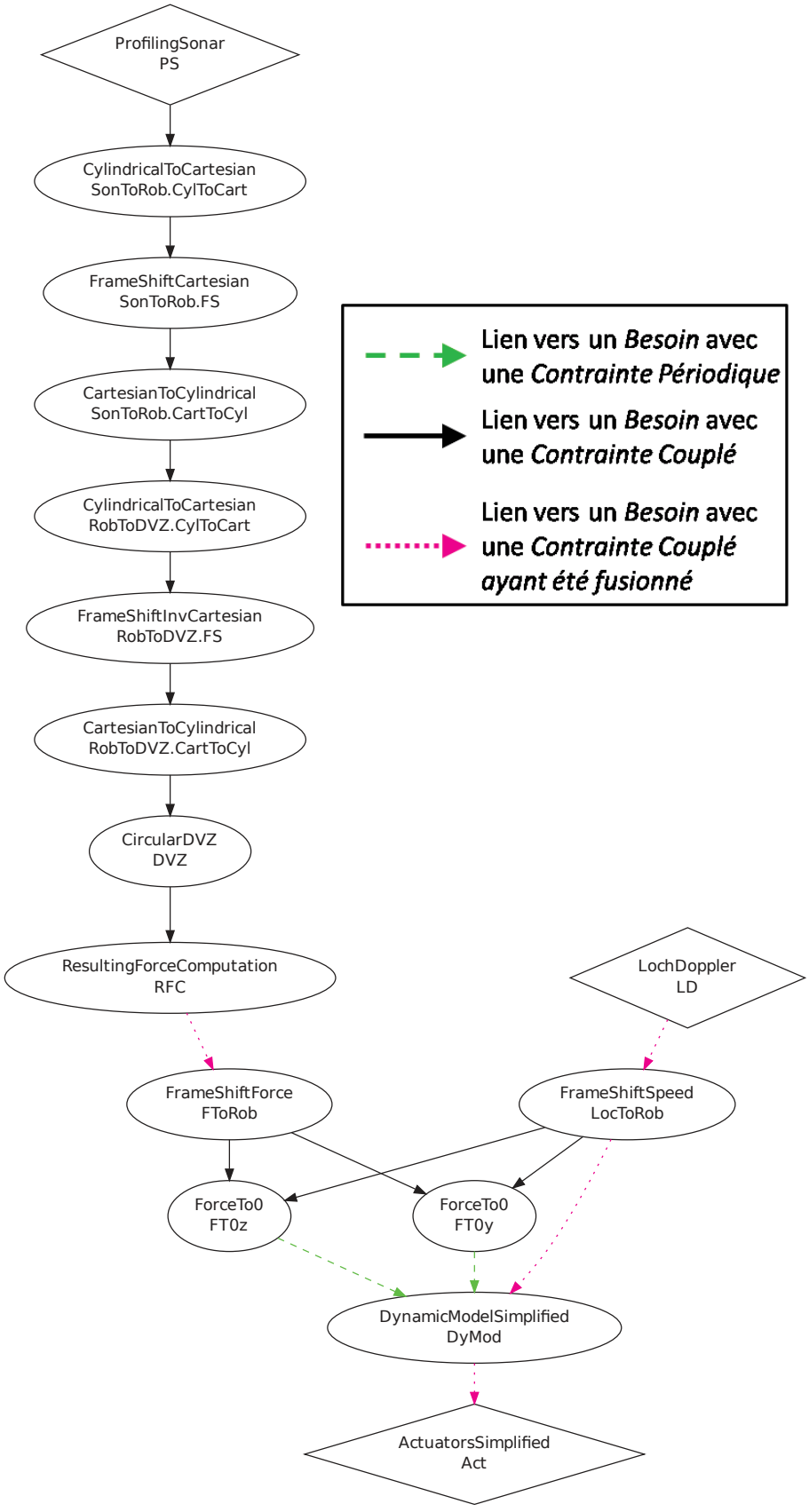

FIGURE 6.8 – Simplification des *Liens* du *Graphe d'Association de Connaissances* exemple

6.3 Diffusion des Contraintes

L'étape suivante consiste à diffuser les *propriétés* de nos *Contraintes* pour en déduire la valeur de deux *propriétés* de chaque graphe temporellement indépendant.

Définition 43:

Soit G un *Graphe d'Association de Connaissances*, les deux *propriétés* de *Contraintes* temporelles portant dessus sont :
$t_{comp_{max}}(G)$, la durée maximale d'exécution des entités du graphe
$T_{exe}(G)$, l'ensemble des périodes possibles pour l'exécution du graphe

Nous souhaitons donc déterminer, pour chaque graphe indépendant, les valeurs de $t_{comp_{max}}$ et T_{exe}.

6.3.1 Structuration du Graphe en Blocs

Ce processus se déroule en deux parties. Premièrement, il faut exprimer les parallélismes et les sérialisations dans notre graphe. Pour cela, nous allons isoler des portions de graphe que nous appellerons *Blocs*. Nous introduisons deux types de *Blocs*, *Blocs* séries et *Blocs* parallèles.

Les *Blocs* sont eux-mêmes des *Graphes d'Association de Connaissances* et possèdent donc les *propriétés* décrites dans la Définition 43. Les *Blocs* peuvent contenir des *Sous-Blocs*. Ceux-ci sont soit des *Atomes* soit d'autres *Blocs*.

Définition 44:

Deux *Sous-Blocs* (ou plus) peuvent être regroupés dans un *Bloc* série si et seulement si une relation d'ordre peut être déduite des *Liens* entre eux ou entre les *Sous-Blocs* les connectant ensemble.

Définition 45:

Deux *Sous-Blocs* (ou plus) peuvent être regroupés dans un *Bloc* parallèle si et seulement si aucune relation d'ordre ne peut être établie entre eux en se basant sur leurs connexions. Cela implique qu'ils ne soient pas connectés ensemble et que leurs connexions amonts (i.e. connexions dirigées vers ces *Sous-Blocs*) et avals (i.e. connexions depuis ces *Sous-Blocs*) soient identiques c'est-à-dire soit liées au même *Sous-Bloc* soit sans connexion.

La Figure 6.9 et la Figure 6.10 présentent des exemples illustrant ces deux définitions.

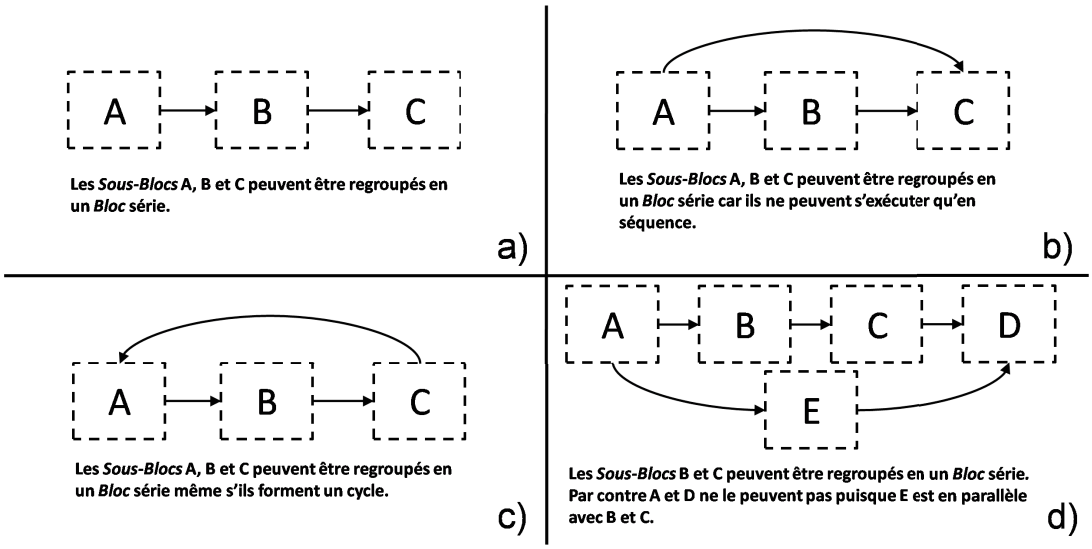

FIGURE 6.9 – Exemples de *Sous-Blocs* pouvant être combinés ou non en un *Bloc* série

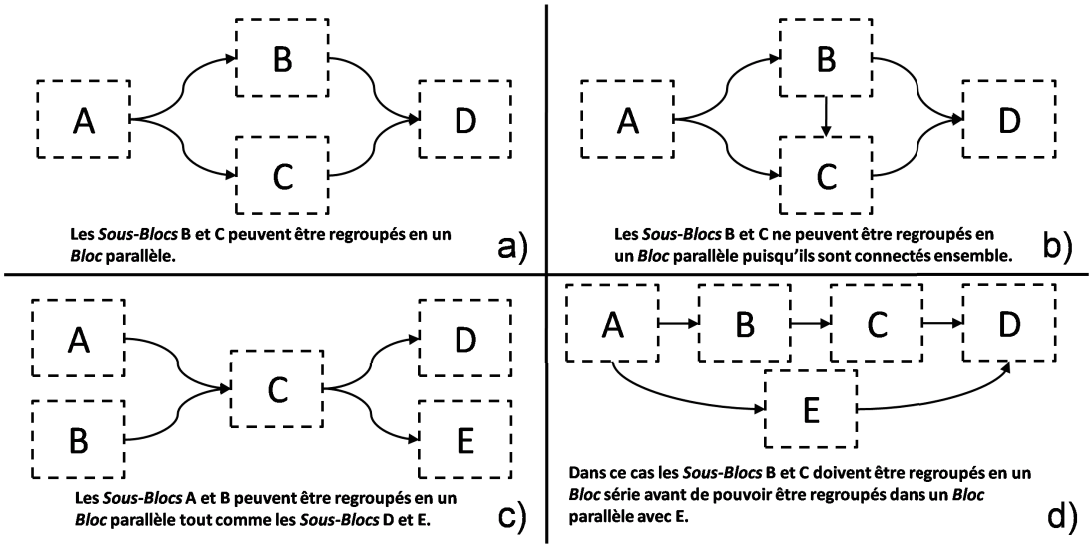

FIGURE 6.10 – Exemples de *Sous-Blocs* pouvant être combinés ou non en un *Bloc* parallèle

Exemple 6.3:

La structuration sous forme de *Blocs* de l'exemple de la Figure 6.8 est présentée Figure 6.11. Ce *Graphe d'Association de Connaissances* peut donc être représenté comme un *Bloc* série $S4$ comprenant trois *Sous-Blocs*, deux parallèles $P1$ et $P2$ et un série $S3$. Le *Bloc* $S3$ contient deux *Atomes* qui sont placés en série puisque directement reliés l'un à l'autre. $P2$ contient les deux *Atomes* d'annulation de force qui sont en parallèle puisqu'étant connectés en amont du modèle dynamique et en aval du changement de repère en force. $P1$ comprend deux *Sous-Blocs* série $S1$ et $S2$.

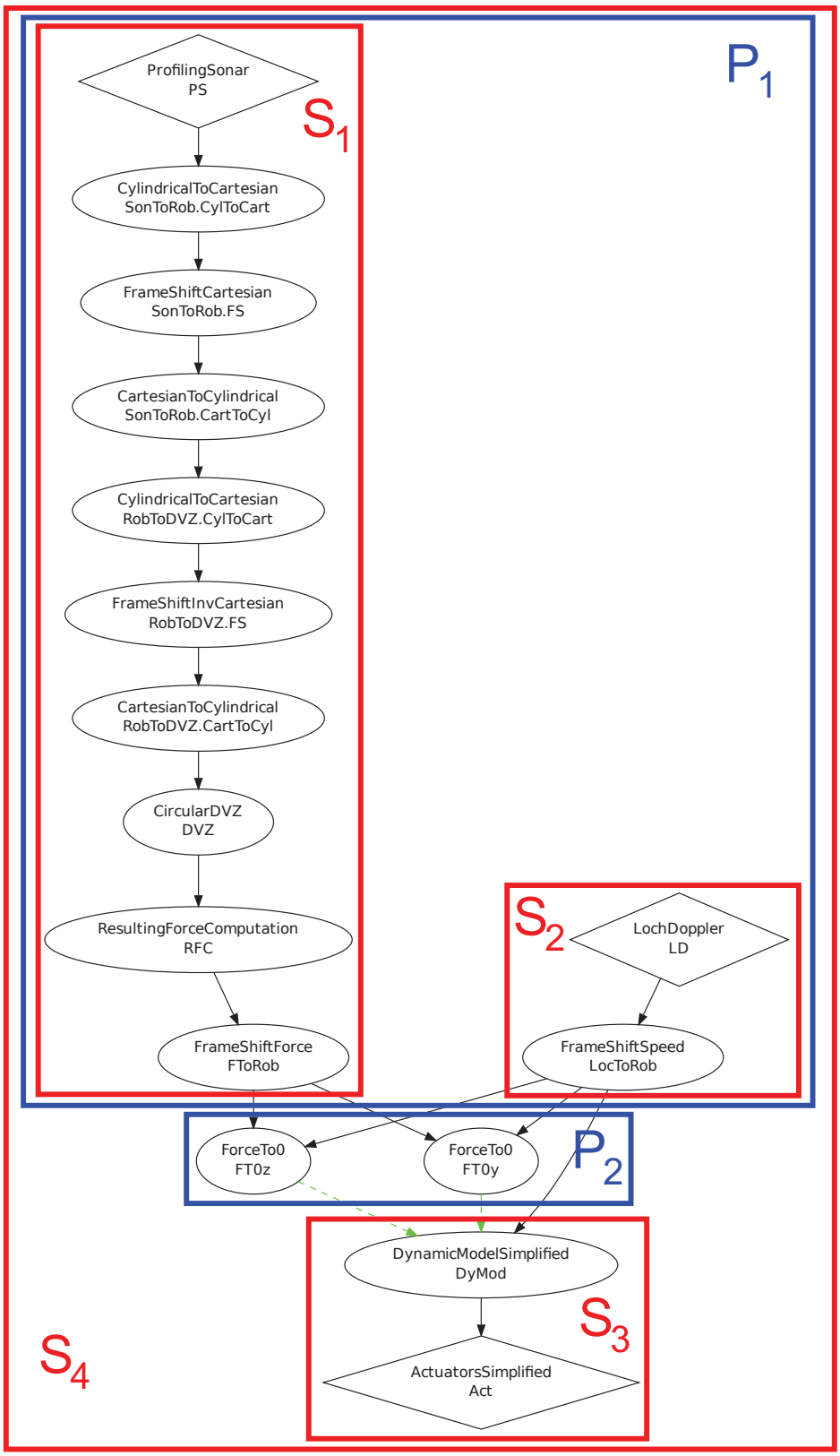

FIGURE 6.11 – Structuration de notre exemple sous forme de *Blocs*

En outre, il est intéressant de noter que même si l'*Instance LocToRob* possède un *Lien* vers le modèle dynamique, sa connexion avec les *Atomes* d'annulation de force impose une relation d'ordre entre eux permettant de placer $S2$ en parallèle avec $S1$. Il faut donc comprendre qu'il n'y a pas, dans notre représentation des *Blocs*, de parallélisme entre un *Bloc* et un *Lien*, comme cela était également illustré Figure 6.9, exemple b).

6.3.2 Fonctions de diffusion des Contraintes

Une fois le graphe structuré par les différents *Blocs*, nous pouvons diffuser les *Contraintes* des *Atomes* à notre *Composition*. Ce processus est réalisé à l'aide de fonctions de diffusion qui dépendent de la nature du *Bloc*. Ces fonctions vont nous permettre de déterminer une expression littérale des différentes *propriétés* des *Contraintes* temporelles. Celles-ci seront ensuite valuées de manière spécifique pour chaque cible d'implémentation.

Considérons tout d'abord un *Bloc* B_s. Il est constitué de n *Blocs*. Parmi eux, les *Sous-Blocs* 1 à i sont eux-mêmes des *Blocs*, le j-ième *Bloc* étant noté B_j tandis que les *Sous-Blocs* $i+1$ à n sont des *Atomes*, le k-ième étant noté At_k. Ce *Bloc* est également connecté à m *Besoins* sur lesquels portent des *Contraintes* temporelles de *nature Périodique* appartenant à d'autres *Blocs* (faisant potentiellement partie du même graphe ou d'un graphe temporellement indépendant). Le o-ième de ces *Besoins* est noté ne_o. Nous devons tout d'abord calculer $t_{comp_{max}}(B_s)$.

S'il s'agit d'un *Bloc* série, cela est fait à l'aide de la fonction (6.1).

$$t_{comp_{max}}(B_s) = \sum_{j=1}^{i} t_{comp_{max}}(B_j) + \sum_{k=i+1}^{n} t_{comp_{max}}(At_k) + t_{comm}(B_s) \tag{6.1}$$

Dans un *Bloc* série où chaque *Sous-Bloc* est calculé l'un à la suite de l'autre, une estimation du temps de calcul total dans le pire cas est égal à la somme des temps de calcul (pire cas) de chaque *Sous-Bloc* pris individuellement auquel est ajouté un terme, t_{comm}, qui réifie la somme totale des temps de transition entre les différentes entités contenues dans le *Bloc*. Ce terme contient aussi bien les temps de transfert des données d'un *Atome* à un autre que les délais induits par l'architecture logicielle (commutation de tâches temps-réel, ordonnancement par exemple).

S'il s'agit d'un *Bloc* parallèle, cela est fait à l'aide de la fonction (6.2).

$$t_{comp_{max}}(B_s) = f(t_{comp_{max}}(B_1), ..., t_{comp_{max}}(B_i), t_{comp_{max}}(At_{i+1}), ..., t_{comp_{max}}(At_n)) + t_{comm}(B_s) \tag{6.2}$$

On retrouve ici le terme t_{comm} qui sert toujours à prendre en compte les temps de transition entre le calcul de chaque entité. Néanmoins contrairement au cas des *Blocs* série, la relation entre le temps de calcul total et les temps de calcul individuels est matérialisée par une fonction f qui ne peut pas être définie à ce stade de l'approche. En effet, cette fonction dépend de la manière dont sont gérés les parallélismes dans l'architecture logicielle et dépend de fait de la cible d'implémentation logicielle. Or, à ce stade, comme expliqué au début du chapitre, le processus est encore indépendant de la cible d'implémentation. La définition de f ne sera donc déterminée que lors de l'étape suivante, la valuation des *propriétés* des *Contraintes*. Nous pouvons néanmoins ici préciser les différentes formes que peut prendre f. Deux définitions principales se dégagent. La première suppose que le parallélisme n'est pas utilisé ou n'est pas utilisable (monoprocesseur), c'est-à-dire que tous les calculs s'effectuent les uns à la suite des autres. La fonction est dans ce cas identique à celle utilisée dans le cas d'un *Bloc* série (équation (6.1). La seconde suppose que les n entités sont toutes exécutées en parallèle (multiprocesseur). Dans ce cas, on a :

$$f(t_{comp_{max}}(B_1), ..., t_{comp_{max}}(B_i), t_{comp_{max}}(At_{i+1}), ..., t_{comp_{max}}(At_n)) = \\ max(t_{comp_{max}}(B_1), ..., t_{comp_{max}}(B_i), t_{comp_{max}}(At_{i+1}), ..., t_{comp_{max}}(At_n)) \quad (6.3)$$

En effet, si le calcul de toutes les entités est effectué en parallèle alors le temps total dépend du temps d'exécution le plus long parmi les entités. Bien entendu, les deux définitions données ici pour f ne sont que des cas "extrêmes". Toute combinaison de ces deux fonctions est bien entendu possible en fonction à la fois de la volonté du concepteur et des possibilités du ou des calculateurs (nombre maximal de calculs parallélisables notamment). Il faut enfin noter que, dans notre cas, le *Middleware* que nous allons utiliser (cf. Chapitre 7) ne permet que d'effectuer les calculs sur un unique processeur. De fait, nous n'utiliserons dans nos travaux que la première fonction (celle pour les entités placées en série).

De là il est possible de déterminer la valeur de $T_{exe}(B_s)$. Cela est fait à l'aide de la fonction (6.4).

$$T_{exe}(B_s) = [t_{comp_{max}}(B_s) \ \infty] \bigcap_{j=1}^{i} T_{exe}(B_j) \bigcap_{k=i+1}^{n} T_{phy}(At_k) \bigcap_{o=1}^{m} T_{Need}(ne_o) \quad (6.4)$$

Le premier terme réifie l'impact de la durée des calculs sur l'ensemble des périodes admissibles. En effet, nous ne pouvons pas réitérer nos calculs plus rapidement que le temps mis pour les effectuer. Néanmoins, étant donné que nous considérons pour cela le pire cas des

temps de calcul, notre approche est clairement très conservative. En effet, individuellement, chaque *Atome* n'a qu'une certaine probabilité de s'exécuter à son pire cas et donc il est peu probable que la *Composition* de ceux-ci s'exécute à chaque cycle avec le pire cas au niveau temporel. En outre, bien entendu suivant l'amplitude et l'occurrence (i.e. nombre de cycles en dépassement sur une fenêtre glissante) des dépassements, un système temps-réel peut, grâce à l'ordonnanceur, parvenir à gérer des dépassements et à retrouver sa régularité d'exécution.

Ensuite, la période d'exécution est obtenue par calcul de l'intersection des périodes d'exécution possibles des *Sous-Blocs* qu'il contient et de leur intersection avec les périodes de rafraichissement désirées pour les *Besoins* ayant des *Contraintes* de *nature Périodique*.

Exemple 6.4:

Appliquons ces formules à l'Exemple 6.3 (Figure 6.11).

On value ainsi les *propriétés* du *Bloc* série englobant notre *Graphe d'Association de Connaissances*, S_4, comme :

$$t_{comp_{max}}(S_4) = t_{comp_{max}}(S_3) + t_{comp_{max}}(P_2) + t_{comp_{max}}(P_1) + t_{comm}(S_4) \quad (6.5)$$

$$T_{exe}(S_4) = [t_{comp_{max}}(S_4)\ \infty] \cap T_{exe}(S_3) \cap T_{exe}(P_2) \cap T_{exe}(P_1) \quad (6.6)$$

Ici, $t_{comm}(S_4)$ englobe les durées de transition entre les *Blocs* P_1, P_2 et S_3. C'est-à-dire la durée d'échange au niveau des *Liens* entre les *Instances* $FToRob$ et $FT0y$, $FToRob$ et $FT0z$, $LocToRob$ et $FT0y$, $LocToRob$ et $FT0z$, $LocToRob$ et $DyMod$, $FT0y$ et $DyMod$ et enfin $FT0z$ et $DyMod$. Ici, vu que nous n'avons qu'un seul *Graphe d'Association de Connaissances* il n'y a pas de *Besoin* de *nature Périodique* à utiliser pour le calcul de $T_{exe}(S_4)$.

Les *propriétés* de S_3 sont obtenues par les équations suivantes car il s'agit d'un *Bloc* série :

$$t_{comp_{max}}(S_3) = t_{comp_{max}}(DyMod) + t_{comp_{max}}(Act) + t_{comm}(S_3) \quad (6.7)$$

$$T_{exe}(S_3) = [t_{comp_{max}}(S_3)\ \infty] \cap T_{phy}(DyMod) \cap T_{phy}(Act) \quad (6.8)$$

$t_{comm}(S_4)$ comprend la durée de transition entre les *Instances* $DyMod$ et Act.

Les *propriétés* de P_2 sont obtenues par les équations suivantes car il s'agit d'un *Bloc* parallèle :

$$t_{comp_{max}}(P_2) = f_{P_2}(t_{comp_{max}}(FT0y), t_{comp_{max}}(FT0z)) + t_{comm}(P_2) \quad (6.9)$$

$$T_{exe}(P_2) = [t_{comp_{max}}(P_2)\ \infty] \cap T_{phy}(FT0y) \cap T_{phy}(FT0z) \cap T_{Need}(DyMod.dv) \cap T_{Need}(DyMod.dw) \quad (6.10)$$

$t_{comm}(P_2)$ intègre les temps de transition entre l'exécution des *Instances* $FT0y$ et $FT0z$. En outre, celles-ci étant connectées à des *Besoins* de *nature Périodique* de l'*Instance* $DyMod$, nous voyons apparaître l'impact de ceux-ci dans le calcul de $T_{exe}(P_2)$.

A titre d'illustration, dans notre cas applicatif (i.e. pas d'utilisation du parallélisme) et en prenant un peu d'avance sur l'étape suivante, la fonction de diffusion (6.9) devient :

$$t_{comp_{max}}(P_2) = t_{comp_{max}}(FT0y) + t_{comp_{max}}(FT0z) + t_{comm}(P_2) \tag{6.11}$$

Les *propriétés* de P_1 sont obtenues par les équations suivantes car il s'agit d'un *Bloc* parallèle :

$$t_{comp_{max}}(P_1) = f_{P_1}(t_{comp_{max}}(S_1), t_{comp_{max}}(S_2)) + t_{comm}(P_1) \tag{6.12}$$

$$T_{exe}(P_1) = [t_{comp_{max}}(P_1)\ \infty] \cap T_{exe}(S_1) \cap T_{exe}(S_2) \tag{6.13}$$

Avec :

$$\begin{aligned} t_{comp_{max}}(S_1) = t_{comp_{max}}(PS) + t_{comp_{max}}(SonToRob.CylToCart) + t_{comp_{max}}(SonToRob.FS) \\ + t_{comp_{max}}(SonToRob.CartToCyl) + t_{comp_{max}}(RobToDVZ.CylToCart) \\ + t_{comp_{max}}(RobToDVZ.FS) + t_{comp_{max}}(RobToDVZ.CartToCyl) + t_{comp_{max}}(DVZ) \\ + t_{comp_{max}}(RFC) + t_{comp_{max}}(FToRob) + t_{comm}(S_1) \end{aligned} \tag{6.14}$$

$$\begin{aligned} T_{exe}(S_1) = [t_{comp_{max}}(S_1)\ \infty] \cap T_{phy}(PS) \cap T_{phy}(SonToRob.CylToCart) \cap T_{phy}(SonToRob.FS) \\ \cap T_{phy}(SonToRob.CartToCyl) \cap T_{phy}(RobToDVZ.CylToCart) \cap T_{phy}(RobToDVZ.FS) \\ \cap T_{phy}(RobToDVZ.CartToCyl) \cap T_{phy}(DVZ) \cap T_{phy}(RFC) \cap T_{phy}(FToRob) \end{aligned} \tag{6.15}$$

$$t_{comp_{max}}(S_2) = t_{comp_{max}}(LD) + t_{comp_{max}}(LocToRob) + t_{comm}(S_2) \tag{6.16}$$

$$T_{exe}(S_2) = [t_{comp_{max}}(S_2)\ \infty] \cap T_{phy}(LD) \cap T_{phy}(LocToRob) \tag{6.17}$$

Puisque S_1 et S_2 sont des *Blocs* série. Ici, $t_{comm}(P_1)$ comprend la durée de transition entre l'exécution des *Blocs* S_1 et S_2 et au sein de ceux-ci, $t_{comm}(S_1)$ et $t_{comm}(S_2)$ représentent les durées de transition entre les différentes *Instances* les constituant.

6.4 Valuation des propriétés et paramétrage de l'implémentation

Lors de l'étape précédente, nous avons déterminé, pour chacun des graphes temporellement indépendant constituant notre *Composition*, l'expression littérale des *propriétés* des *Contraintes* temporelles. Cette expression est déterminée de manière indépendante des spécificités de la cible d'implémentation. L'étape de valuation des *propriétés* a pour objectif de les valuer dans le cas d'une implémentation spécifique afin de permettre la poursuite des étapes suivantes.

Lors de cette étape, il est nécessaire de fixer :

- La fonction de diffusion choisie pour chaque *Bloc* parallèle.
- Les valeurs de $t_{comp_{max}}$ et de T_{phy} pour chaque *Atome* avec des *Contraintes* temporelles de *nature Périodique*.
- Les valeurs de $t_{comp_{max}}$ pour chaque *Atome* avec des *Contraintes* temporelles de *nature Sporadique* et de $t_{exe_{max}}$ pour chaque *Atome* utilisé dans une *Entité Composable* de jonction (entités qui ont toutes nécessairement des *Contraintes* temporelles de *nature Sporadique*). En effet, la phase de commutation d'un ensemble de connaissances à un autre entraîne une perte momentanée du contrôle du robot. Dès lors, il est nécessaire de s'assurer que cette perte de contrôle respecte les *Contraintes* fixées afin que la commutation ne mette pas en danger l'intégrité du robot et de son environnement. Si une *Entité Composable* de jonction contient plusieurs *Atomes*, alors la valeur de $t_{comp_{max}}$ totale est établie à l'aide de la fonction de diffusion (6.1) et les valeurs de $t_{exe_{max}}$ doivent être identiques pour chaque *Atome*.
- Les valeurs de t_{comm} pour chaque *Bloc*. Comme dit précédemment, ce temps doit prendre en compte à la fois la durée d'échange de données entre *Atomes* au niveau des *Liens* ainsi que les délais induits par les mécanismes du *Middleware*.

Une fois ces différentes *propriétés* valuées, nous pouvons appliquer les formules déterminées à l'étape précédente. Elles vont nous permettre de déterminer l'ensemble des périodes d'exécution possibles pour chaque graphe temporellement indépendant. Il est tout d'abord, à partir de ces informations, nécessaire de vérifier si la *Composition* est implémentable.

Définition 46:

Soit une *Composition* c représentée par un *Graphe d'Association de Connaissances*, divisé en n graphes temporellement indépendants, $G_1 ... G_n$. Notons $E_{junc_{Compo}}$ l'ensemble des *Entités Composables* de jonction contenues dans toutes les *Alternatives* de la *Composition*. c est implémentable, noté $c \in C_I$, avec C_I l'ensemble des *Compositions* implémentables si et seulement si :
$T_{exe}(G_i) \neq \emptyset \ \forall \ i \in [1 \ .. \ n]$ (Tous les graphes temporellement indépendants ont une période d'exécution possible.)
$t_{comp_{max}}(e_{junc}) \leq t_{exe_{max}}(e_{junc}) \ \forall \ e \in E_{junc_{Compo}}$ (Toutes les *Entités Composables* de jonction ne doivent pas prendre plus de temps pour réaliser leur commutation que le temps maximal admissible pour des raisons de sécurité ou de stabilité).

Si la *Composition* n'est pas implémentable, l'étude des *Contraintes* va nous fournir des informations clés afin d'analyser les causes de la non-implémentabilité et ainsi permettre de guider les concepteurs de l'architecture de contrôle vers les modifications de commande à apporter et/ou les concepteurs de l'architecture matérielle et logicielle du robot vers les changements de technologie nécessaires.

Si la *Composition* est implémentable, il est alors nécessaire de choisir la période d'exécution de chaque graphe et donc des *Instances* qu'il contient. Il est à noter que dans le cas où une *Instance* appartiendrait à plusieurs graphes alternatifs, nous conseillons, dans la mesure où cela respecte nos *Contraintes* temporelles, de choisir des périodes identiques pour chaque graphe alternatif, afin de réduire les besoins de changement de période qui sont parfois difficiles à mettre en œuvre suivant le fonctionnement du *Middleware*. Toutefois, il ne faut pas négliger l'importance des intervalles de période déterminés dans le cadre de stratégies de contrôle de l'exécution comme les approches issues de l'*autonomic computing*.

6.5 Implémentation de cette approche

Avant de s'intéresser, dans le chapitre suivant, à la projection de notre *Composition* sur un *Middleware*, nous devons mentionner brièvement l'implémentation des différentes étapes de la phase d'étude des *Contraintes*. Celle-ci demeure pour le moment limitée et mise en œuvre dans une librairie logicielle indépendante de celle présentée à l'Annexe D.

Même si, en l'état actuel, elle demeure limitée dans les fonctionnalités implémentées, elle propose une bonne base pour de futurs développements. Notre librairie propose une structure

de données permettant de représenter notre graphe comme une liste de nœuds et d'arcs. Il est à noter que les *Contraintes* temporelles portant sur les *Besoins* de nos entités sont ici reportées sur les *Liens* pour un souci de simplicité et de représentation. Partant de cette structure de données, nous avons implémenté l'étape de vérification de compatibilité des *Contraintes* temporelles ainsi que certaines étapes de la structuration du graphe, à savoir la simplification de celui-ci (suppression des entités non périodiques, fusion des liens) et la séparation en différents graphes sur la base des *Contraintes*. A chaque étape du processus, un ou plusieurs fichiers en langage *DOT* [GKN15] sont générés afin de pouvoir automatiquement produire une représentation visuelle de notre graphe en utilisant l'outil *dot* permettant de dessiner des graphes orientés.

Néanmoins, cette librairie présente encore un grand nombre de limitations. Premièrement, nous n'avons pas encore défini de description générique des *Compositions*. Dès lors, le *Graphe d'Association de Connaissances* associé à celle-ci doit être défini manuellement. En outre, l'absence actuelle d'interface entre cette librairie et celle chargée de gérer la description des *Entités Composables* ne nous permet pas à la fois de vérifier la validité du graphe (notamment si les *Besoins* et *Produits* utilisés pour décrire les *Liens* appartiennent effectivement aux entités liées) et de mettre en place la phase chargée de transformer le graphe pour qu'il ne soit plus constitué que d'*Atomes*. De fait, même s'il est possible de décrire un *Graphe d'Association de Connaissances* qui ne soit pas constitué entièrement d'*Instances* d'*Atomes*, il est nécessaire d'effectuer manuellement la conversion vers un *Graphe d'Association de Connaissances* ne contenant que des *Instances* d'*Atomes* afin de pouvoir appliquer les étapes déjà implémentées.

Enfin, la structuration des graphes sous forme de *Blocs* n'est pas non plus implémentée. La détection des parallélismes et sérialisations dans un graphe orienté est en plus une problématique complexe. Néanmoins, les travaux sur l'ordonnancement multi-cœurs tels que [SACG11, Section VI et Figure 3 notamment] peuvent offrir des pistes intéressantes d'algorithmes pour effectuer la structuration en *Blocs* de nos *Graphes d'Association de Connaissances*. Néanmoins, ces travaux se limitent souvent au cas où les graphes sont acycliques ce qui est rarement le cas d'une *Composition* utilisée pour décrire le contrôleur d'un robot.

6.6 Points clés du chapitre

- La phase d'étude des *Contraintes* a pour but de vérifier l'implémentabilité du contrôle décrit sur une cible technologique et logicielle donnée et de pouvoir déterminer les valeurs des différents paramètres de l'implémentation et notamment la période d'exécution des entités ayant une exécution périodique.
- Cette phase commence par une restructuration du *Graphe d'Association de Connaissances* associé à la *Composition* afin d'obtenir un ensemble de graphes temporellement indépendants et constitués uniquement des *Instances* d'*Atomes* ayant des *Contraintes* de *nature Périodique* qui doivent toutes s'exécuter à la même période.
- Il faut ensuite, pour chaque graphe, exprimer les sérialisations et les parallélismes dans l'ordre des calculs à effectuer.
- Nous pouvons dès lors appliquer les formules de diffusion des *Contraintes* qui vont nous permettre de déterminer l'expression littérale de l'ensemble des périodes d'exécution possibles de chaque graphe en fonction des périodes d'exécution possibles des entités qu'il contient.
- Cet ensemble de périodes d'exécution possibles correspond à l'ensemble des périodes pour lesquelles la stabilité du contrôle est garantie tout en respectant les contraintes inhérentes à la cible d'implémentation.
- Il faut ensuite valuer les *propriétés* des *Contraintes* portant sur les différentes entités et appliquer les formules de calcul des périodes d'exécution précédemment établies pour vérifier l'implémentabilité de la *Composition*.
- Si la *Composition* n'est pas implémentable, l'étude des *Contraintes* va fournir aux concepteurs des architectures matérielles, logicielles et de contrôle des informations essentielles pour les aider à déterminer les modifications à mettre en œuvre pour rendre la *Composition* implémentable.
- Si elle est implémentable, alors nous pouvons choisir, parmi l'ensemble des périodes d'exécution possibles, celle que nous souhaitons affecter à chaque partie de la *Composition*.

Chapitre 7

Projection sur un Middleware, l'exemple de ContrACT

Au chapitre précédent, nous avons vu comment déterminer les périodes d'exécution de nos *Atomes* qui ont des *Contraintes* périodiques. A partir des *Contraintes*, nous devons désormais déterminer comment utiliser les différentes connaissances dans notre logiciel de contrôle. Il nous faut donc définir un ensemble de règles qui, pour une architecture logicielle et un *Middleware* donnés, vont nous permettre de savoir comment tel ou tel *Atome* sera utilisé.

Notre description étant indépendante des différentes cibles d'implémentation logicielles avec lesquelles elle est susceptible d'être implémentée, il nous faut établir un ensemble de règles pratiques pour chacune d'entre elles en fonction de ses spécificités. Ces règles, qui sont construites grâce à l'expérience du concepteur de l'architecture logicielle, vont nous permettre de déterminer les mécanismes à utiliser suivant les différentes *natures* de *Contraintes* et surtout ce qu'il ne faut pas faire car les mécanismes fournis ne le permettent pas.

Comme les règles sont spécifiques à un *Middleware* cible donné, nous allons donc illustrer notre approche avec l'exemple du *Middleware* ContrACT. Nous allons commencer par présenter les spécificités du *Middleware* ContrACT puis nous détaillerons les règles qui découlent de celles-ci.

7.1 ContrACT

Développée au LIRMM, l'approche ContrACT est une méthodologie de développement d'architectures de contrôle associée à un *Middleware* permettant leur mise en œuvre suivant les concepts établis par la méthodologie. Elle vise à permettre de respecter des considérations clés du génie logiciel [Pas10] :

– La conception de briques logicielles indépendantes permettant un maximum de réutili-

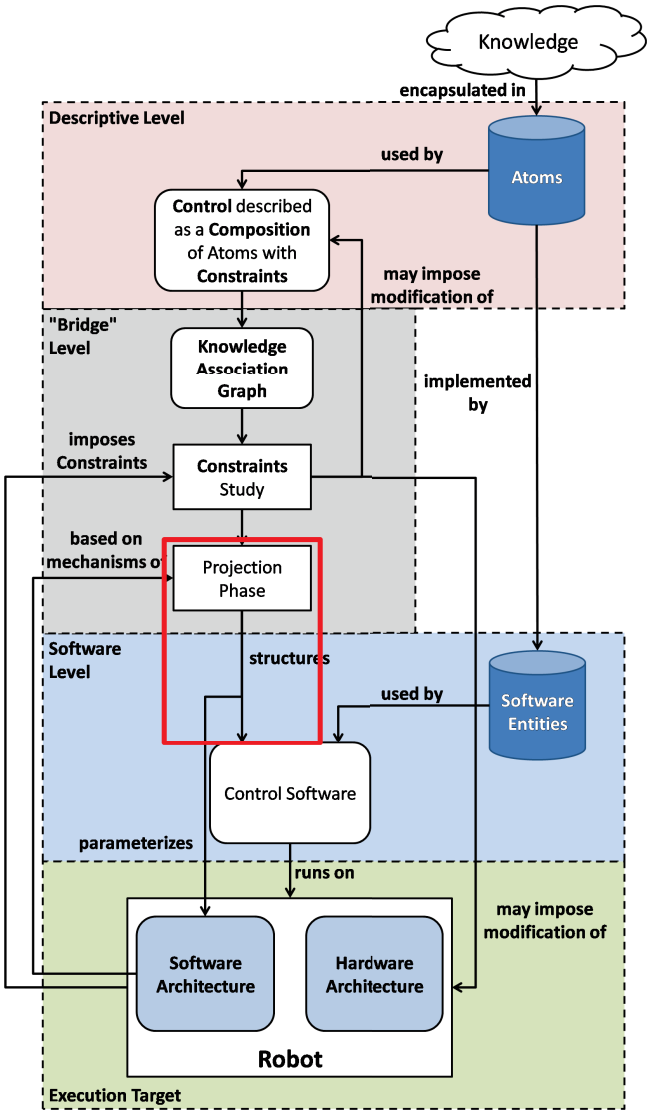

FIGURE 7.1 – Aspects de la méthodologie abordés dans le chapitre 7

sabilité des composants logiciels.

– Une composition flexible de ces entités logicielles favorisant l'évolutivité du système.

Les briques logicielles utilisées dans ContrACT sont appelées modules. A l'instar d'objets ou de nos *Atomes*, les modules comprennent un comportement interne, non accessible directement depuis l'extérieur et une interface qui leur permet d'échanger des informations entre modules ou de paramétrer leur comportement. Le comportement interne d'un module peut être divisé en deux parties, d'un côté un code commun générique (sous forme d'un squelette de code) chargé de la gestion du module (réception des messages, appel au code utilisateur, initialisation, fermeture) et de l'autre le code spécifique à l'utilisateur dont la structure dépend

du rôle joué ou de la fonctionnalité assurée par le module. L'interface de tous les modules, basée sur des ports typés, est organisée de manière identique, comme montré à la Figure 7.2.

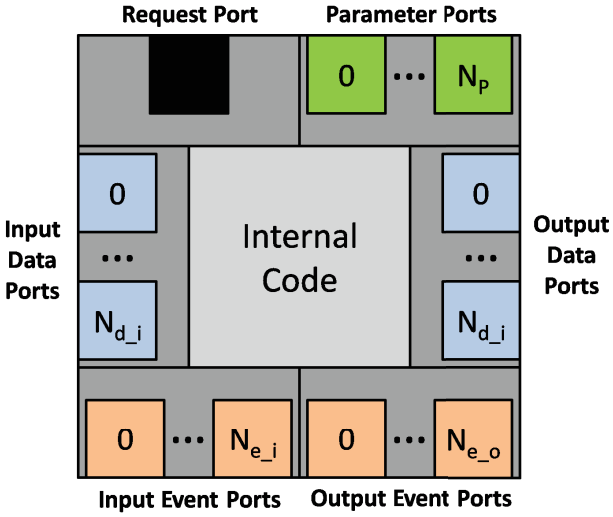

FIGURE 7.2 – Structure d'un module

Cette interface permet l'assemblage des modules par interconnexion de ports. Les échanges de données entre modules relèvent du modèle producteur/consommateur (*publisher/subscriber*).

Des ports de paramètres permettent à un autre module de valuer certaines données internes aux modules. Les ports d'évènements permettent aux modules d'émettre ou de recevoir des évènements, c'est-à-dire des données envoyées de manière sporadique. La connexion entre deux ports d'évènements peut soit se faire de manière unique (le receveur se désabonne de l'émetteur après réception de l'évènement) soit de manière "continue" (i.e. le receveur récupèrera tout les évènements émis), on parlera alors de flux d'évènements. Des ports de données permettent d'échanger des informations de manière périodique entre les modules. La connexion entre un port d'entrée d'un module et un port de sortie d'un autre module est appelée flux de données. En outre, les modules possèdent un port de requête qui permet de gérer leur activité (démarrage/arrêt) et leurs connexions (abonnement/désabonnement). Du point de vue logiciel, chacun de ces ports est une boîte aux lettres qui possède son propre buffer et peut donc stocker quelques données. Enfin, il est à noter que, quel que soit le type de flux (données ou évènements), c'est toujours le module consommateur de l'information qui doit réaliser les actions d'abonnement (connexion au flux) ou de désabonnement (déconnexion du flux).

De fait, la gestion des flux entre les modules peut être dynamique, c'est-à-dire qu'à tout moment un flux peut être mis en place ou interrompu. Cela offre une grande flexibilité dans la construction de l'application mais nécessite également une grande précaution dans la manipulation de ces flux. En effet, leur gestion étant totalement découplée, par exemple, du démarrage

ou de l'arrêt d'un module, il faut toujours s'assurer de la bonne interruption des flux dirigés vers un module avant de désactiver celui-ci. Sinon il ne relèvera plus les messages arrivant dans ses boîtes aux lettres ce qui entraînera le débordement de celles-ci et l'apparition de nombreuses erreurs dans l'application.

Les modules sont organisés en deux couches, décisionnelle et exécutive, qui traduisent leur rôle dans l'architecture logicielle ainsi que leurs modalités de fonctionnement (i.e. priorité, mécanismes de mises en œuvre, éléments d'interface utilisables). Ces différentes couches et les relations entre modules sont présentées à la Figure 7.3.

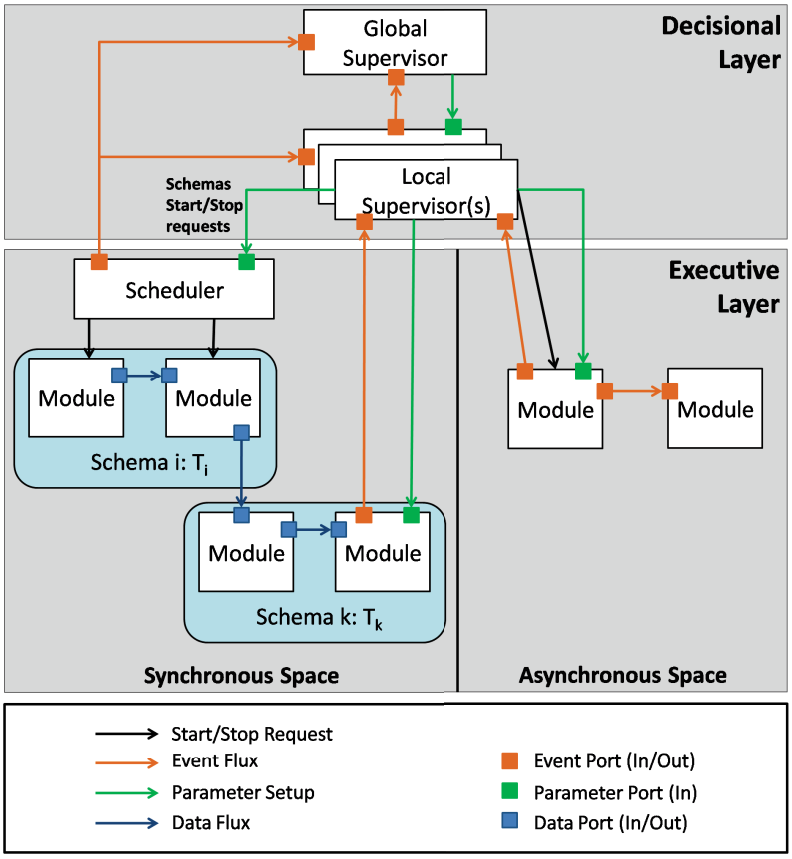

FIGURE 7.3 – Structuration des différentes couches du *Middleware* ContrACT

Dans la couche décisionnelle, les modules sont appelés superviseurs et sont organisés, par le concepteur, de manière hiérarchique suivant le caractère plus ou moins global des décisions qu'ils doivent prendre. Les superviseurs peuvent interagir soit avec les superviseurs de niveau hiérarchique directement supérieur en leur transmettant des notifications sous forme d'évènements soit avec les superviseurs de niveau inférieur ou avec les modules de la couche exécutive en leur transmettant des requêtes et/ou des informations via des paramètres. Ces derniers traduisent la demande émanant d'un superviseur de mise en œuvre d'une décision par

d'autres superviseurs ou par des modules de la couche exécutive. Les superviseurs sont par définition asynchrones car ils ne s'exécutent que lorsqu'un évènement significatif (notification, timer, paramétrage) se produit. De fait, ils n'utilisent pas de port de données.

Leur code utilisateur est structuré en fonctions de supervision qui regroupent un ensemble de règles. Ces fonctions peuvent être activées soit par l'utilisateur (typiquement pour démarrer l'application ou activer un mode de fonctionnement spécifique), soit par des superviseurs de niveau hiérarchique plus élevé. L'activation ou non des fonctions de supervision permet ainsi de sélectionner les ensembles de règles activables, c'est-à-dire qui peuvent s'exécuter et dont les conditions d'activation et d'arrêt sont monitorées. Les règles, pouvant s'exécuter de manière concurrente au sein d'une fonction, sont construites en trois parties :

- Une précondition qui définit les conditions à remplir pour qu'une règle activable soit activée.
- Des actions à effectuer lorsque la règle devient active, il peut s'agir de calculs (courts), de paramétrages, d'envois d'évènements ou d'actions structurantes (activation d'un schéma, i.e. d'un assemblage de modules, et des modules associés, établissement d'un flux entre deux modules).
- Une postcondition qui définit les conditions à remplir pour qu'une règle active soit désactivée. Lorsque la règle devient inactive, les actions structurantes sont annulées (arrêt du schéma, interruption des flux).

La couche exécutive contient les modules qui sont chargés de mettre en œuvre la (ou les) commande(s) du robot choisie(s) par l'étage décisionnel. Ces modules doivent gérer à la fois, l'interaction avec le monde extérieur au contrôleur (capteurs, communications, actionneurs) et la (ou les) commande(s) elle(s)-même(s). Cette couche est divisée en deux domaines : le domaine synchrone qui comprend tous les modules s'exécutant de manière périodique et le domaine asynchrone comprenant tous les modules s'exécutant de manière évènementielle.

Les modules asynchrones ne sont pas ordonnancés et vont s'exécuter sur demande des superviseurs. En outre, ils ne possèdent pas de ports de données. Ils sont divisés en deux catégories. Les modules d'interaction sont utilisés pour implémenter des calculs courts ou des échanges avec des périphériques non périodiques. Ils peuvent être reliés ensemble par des flux d'évènements. Il s'agit des modules les plus prioritaires de la couche exécutive. D'un autre côté, les modules "temps-restant" sont utilisés pour effectuer des calculs longs souvent liés à la gestion de la mission et que ne peuvent réaliser les superviseurs pour des questions de réactivité, ou même des actions dont la durée exacte n'est pas prévisible. Il s'agit des modules

les moins prioritaires au sein du *Middleware*.

Le domaine synchrone comprend tous les modules s'exécutant de manière périodique ainsi que le module ordonnanceur chargé de gérer leur exécution. En effet, la gestion des modules synchrones est assurée par un module ordonnanceur qui pilote l'ordonnanceur du système d'exploitation. Il permet ainsi un contrôle plus précis de leur exécution.

A l'intérieur des modules synchrones, le code utilisateur est organisé autour de trois fonctions :

- Initialisation : les actions à réaliser à la création du module.
- Comportement : les actions à exécuter à chaque exécution du module.
- Terminaison : les actions à réaliser quand le module est détruit.

De plus, lors de la description du module, sa durée critique doit être renseignée. Celle-ci correspond à la durée d'exécution laissée au module par l'ordonnanceur. Si le module dépasse cette durée, l'ordonnanceur commence par lui laisser une durée supplémentaire (ce qui décale d'autant l'ordonnancement des autres modules) et si le module n'a toujours pas terminé son exécution, l'ordonnanceur va réduire sa priorité afin de permettre aux modules suivants de s'exécuter pour respecter leurs échéances, puis le module pourra reprendre son exécution afin de la terminer une fois que tous les autres modules se seront exécutés. Nous pouvons donc constater l'importance d'un réglage précis de cette durée critique. De plus, différentes stratégies de gestion des dépassements peuvent être mises en œuvre dans l'ordonnancement en fonction du nombre et de la récurrence de ceux-ci (par exemple le nombre de dépassements sur une fenêtre glissante de n exécutions).

Les modules sont ensuite regroupés en schémas comme illustré Figure 7.4. On définit un schéma comme un ensemble de modules fonctionnant à une même période. Au sein d'un schéma, les modules sont reliés ensemble par des flux de données. Nous pouvons ainsi définir des contraintes de précédence entre les modules d'un même schéma qui vont permettre à l'ordonnanceur de déterminer dans quel ordre doivent s'exécuter les différents modules du schéma. Il est bien entendu possible d'utiliser plusieurs schémas simultanément, fonctionnant à des fréquences différentes ou non, et de connecter des modules appartenant à différents schémas. Néanmoins, deux points clés sont à noter. Premièrement, il est impossible d'imposer des contraintes de précédence entre des modules appartenant à différents schémas. De fait, les modules connectés ensemble par des flux interschémas doivent être "calculatoirement" indépendants pour éviter les problèmes liés au non-respect des contraintes de précédence. Deuxièmement, il faut s'assurer de correctement paramétrer la fréquence des émissions sur les flux de données. En effet, ContrACT permet d'indiquer le nombre de cycles séparant chaque envoi. Ainsi, si par exemple on connecte la sortie d'un module faisant partie d'un schéma

fonctionnant à une période de 50ms et l'entrée d'un module dans un schéma fonctionnant à 100ms, il faudrait indiquer un envoi tous les 2 cycles bien qu'un envoi à chaque cycle reste possible si besoin est.

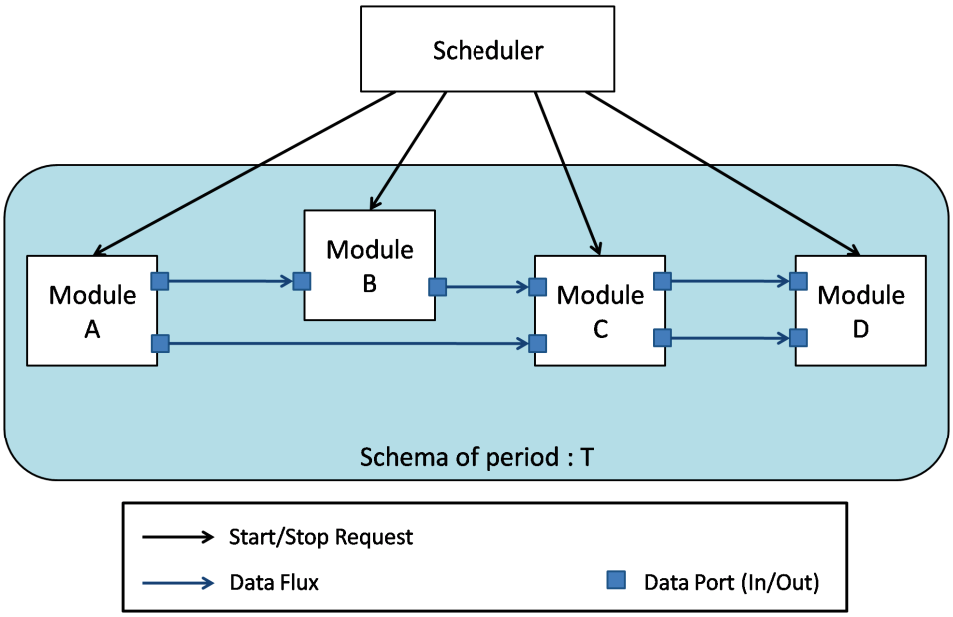

FIGURE 7.4 – Structure d'un schéma ContrACT

Il est à noter que dans le cadre d'interconnexions telles que celles reliant les modules A, B et C dans la Figure 7.4, il est nécessaire de s'assurer que ces modules se trouvent dans le même schéma (s'ils ont la même période d'exécution). En effet, si ce n'est pas le cas, on ne peut plus garantir le respect des contraintes de précédence entre ceux-ci. Ainsi, on peut se retrouver avec un ordre d'exécution où A est le premier à s'exécuter puis C s'exécute et ensuite B. Dans ce cas, les entrées de C ne proviennent pas du même cycle et le calcul effectué par ce module peut être incorrect.

7.2 Règles de projection

Nous allons maintenant proposer un ensemble de règles permettant de guider le concepteur du logiciel de contrôle lors de l'implémentation de nos *Compositions* pour un *Middleware* particulier (i.e. les règles d'implémentation dépendent du *Middleware*). Il s'agit de règles "d'expérience" basées sur les spécificités du *Middleware* et de l'architecture logicielle. Celles-ci ont pour objectif d'indiquer au concepteur quels mécanismes utiliser mais aussi, et surtout, quelles erreurs éviter lors de la mise en œuvre des différentes entités logicielles utilisées.

Dans le cadre de ContrACT, différentes informations sont à obtenir à l'issue de cette phase de projection :

- La répartition de nos différentes entités entre les modules. Nos *Entités Composables* sont encapsulées dans les modules.
- La manière dont les modules synchrones sont regroupés ensemble dans des schémas.
- Les périodes d'exécution de nos différents schémas.
- La durée critique allouée à chaque module.

7.2.1 Entités dont l'exécution a une Contrainte "Constant"

Règle 1:

Tout *Atome* (*Entité Composable*), dont la *Physique* a une *Contrainte* temporelle de *nature Constant*, doit être utilisée dans la **fonction d'initialisation** du ou des modules contenant les entités auxquelles il est connecté.

Ce choix, s'il permet de simplifier la conception de l'initialisation de notre application, a néanmoins l'inconvénient de rendre nécessaire la duplication de certains calculs (un *Atome* par fonction d'initilisation qui est spécifique à chaque module), ce qui rallonge sa durée et augmente les risques d'erreurs. Néanmoins, ceux-ci ne sont effectués qu'une seule fois ce qui limite l'impact temporel de la duplication. D'autres options peuvent être envisagées comme une "phase d'initialisation" utilisant des modules dédiés, pour initialiser les données des autres modules. Sous ContrACT, cela nécessite de développer les modules spécifiques à cette phase. Cela nécessiterait donc un travail de développement coûteux d'autant plus qu'ils sont en partie spécifiques à un logiciel de contrôle donné et donc devraient être redéveloppés pour chaque application.

7.2.2 Entités dont l'exécution a une Contrainte "Sporadique"

Règle 2:

Si un *Atome*, dont la *Physique* a une *Contrainte* temporelle de *nature Sporadique*, possède des *Besoins* avec une *Contrainte Périodique*, il sera implémenté dans un **module synchrone**. Il sera toujours seul dans un tel module.

En effet, le fait pour un tel *Atome* de posséder des *Besoins Périodique* implique que l'*Entité Composable* chargée de vérifier si les conditions d'exécution de sa *Physique* sont réunies, $C_{S_{phy}}$ (voir Tableau 4.2) doit elle-même être réévaluée périodiquement. L'utiliser dans un module synchrone permet donc de s'assurer, via l'ordonnanceur, que cette entité s'exécutera en respectant les *Contraintes* souhaitées.

Règle 3:

Si un *Atome*, dont la *Physique* a une *Contrainte* temporelle de *nature Sporadique*, ne possède pas de *Besoin* avec une *Contrainte Périodique* alors il sera mis en œuvre dans un **module asynchrone**. Le type de module asynchrone (temps-restant ou d'interaction) est laissé à la libre appréciation du concepteur suivant son estimation de la lourdeur et de l'importance des calculs. Plusieurs *Atomes Sporadiques* peuvent être regroupés dans un même module asynchrone.

Règle 4:

Un *Atome*, dont la *Physique* a une *Contrainte* temporelle de *nature Sporadique*, sera implémenté comme présenté à l'Exemple de Code 7.1.

Exemple de Code 7.1 – Structure de l'utilisation d'un Atome Sporadique nommé *a* dont les conditions d'exécution sont représentées par un atome *C*

```
1 //Mise a jour des valeurs des Besoins de a et C
2
3 C.physics() //Evaluation des conditions d'execution
4 if(C.get_product_run().toNoUnit()) // Si les conditions d'execution sont verifiees
5 {
6    a.physics();
7 }
8
9 // Envoi des Produits
```

Dans le seul cas où $C_{S_{phy}} = "TRUE"$, on pourra alors se passer de l'évaluation de la *Physique*. Ce sera souvent le cas si l'*Atome* n'a pas de *Besoin* (notamment s'il représente une interaction avec des entités externes au robot, voir Exemple 5.7) ou n'a que des *Besoins* de *nature Sporadique*.

7.2.3 Connaissances Externes

Règle 5:

Les *Connaissances Externes* seront toujours implémentées dans un **module dédié**.

Cela s'explique par le fait que l'implémentation d'une *Connaissance Externe* est susceptible de différer d'une cible d'implémentation à une autre. Dès lors, dans un souci de modularité,

il est préférable de les isoler des autres connaissances en les plaçant dans des modules à part. Cela apportera plus de modularité au logiciel de contrôle puisqu'un changement technologique ne se répercutera que par un changement ciblé de modules (en plus, bien entendu, d'éventuelles modifications des *Contraintes* et périodes d'exécution).

7.2.4 Entités dont l'exécution a une Contrainte "Périodique"

Règle 6:

Des *Atomes* non couplés temporellement (i.e. non reliés par un lien de couplage) ne pourront **pas faire partie du même module** même s'ils ont la même période d'exécution.

En effet, le fait qu'ils soient découplés temporellement signifie qu'il y a une forte probabilité qu'ils n'aient pas la même période d'exécution dans certaines applications. Or, un module ne pouvant avoir qu'une seule période d'exécution, celle du schéma auquel il appartient, il faut donc les placer dans des modules différents.

Règle 7:

Deux *Atomes* dont l'un serait situé en amont d'un groupe d'*Atomes* appartenant à un module et l'autre serait situé en aval de ce même groupe d'*Atomes*, comme illustré Figure 7.5, ne peuvent **pas faire partie du même module**.

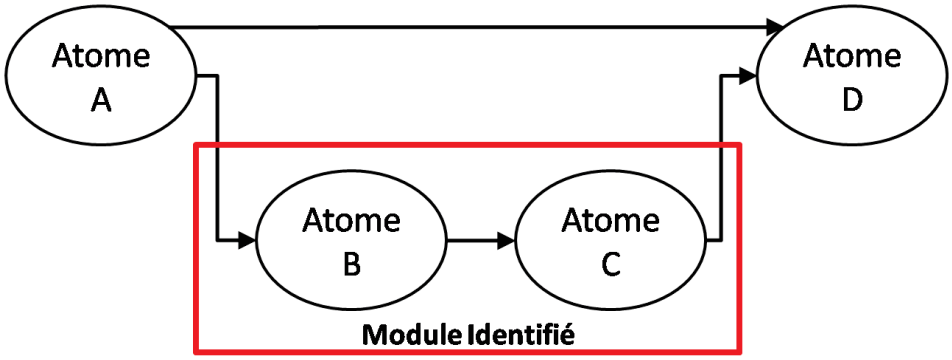

FIGURE 7.5 – Illustration de la Règle 7

Ce choix s'explique par le fait que, si l'on regroupe les *Atomes* A et D dans le même module, alors D s'exécutera forcément immédiatement après A et les contraintes de précédence qui imposent l'ordre A,B,C,D ne pourront plus être respectées.

Règle 8:

Des modules s'exécutant à la même période appartiendront au **même schéma** même si les *Instances* qu'ils contiennent ne sont pas couplées temporellement.

A l'origine, les modules contenant des *Instances* indépendantes temporellement devaient être placés dans des schémas différents pour offrir une plus grande modularité dans les changements de période. Néanmoins, comme à ce stade nous ne pouvons pas ordonnancer les schémas entre eux avec ContrACT, cela entraîne un non respect des *Contraintes* de précédence et cette solution n'a donc pas été retenue.

7.2.5 Alternatives

Règle 9:

Les *Atomes* appartenant à une *Alternative* seront toujours mis dans des **modules séparés** des *Atomes* externes à l'*Alternative*.

Originellement, vu que les *Alternatives* permettent de sélectionner les *Atomes* utilisés en fonction de la situation, il était prévu que les *Atomes* contenus dans les différentes entités substituables d'une *Alternative* appartiennent à des schémas dédiés afin de permettre leur substitution à l'exécution par un arrêt/lancement de schémas. Néanmoins, comme il est à ce stade impossible de faire porter des contraintes de précédence entre schémas, nous perdons alors le déterminisme dans l'ordonnancement. De fait, nous nous sommes limités à des changements de modules, ce qui oblige à repenser la manière de gérer les commutations.

Règle 10:

Au sein d'une *Alternative*, en plus de l'application des règles précédentes (Règles 1 à 6), il faut noter qu'un *Atome* appartenant à une des entités substituables ne pourra pas être mis dans le même module qu'un *Atome* appartenant à une autre entité substituable. Par contre si plusieurs *Atomes* se retrouvent entre différentes entités substituables, il pourront être mis (en accord avec les Règles 1 à 6) dans le même module.

La Figure 7.6 illustre cette Règle au travers d'un exemple théorique.

Ainsi on voit que les *Atomes* B et C sont tous deux utilisés par les entités E_1, E_2 et E_3 et peuvent dès lors être regroupés au sein du même module. Les *Atomes* D et E sont utilisés tous deux dans E_1 et E_2 donc peuvent être regroupés dans le même module mais ne peuvent appartenir au même module que B et C puisqu'ils ne sont pas utilisés dans E_3. H et J sont

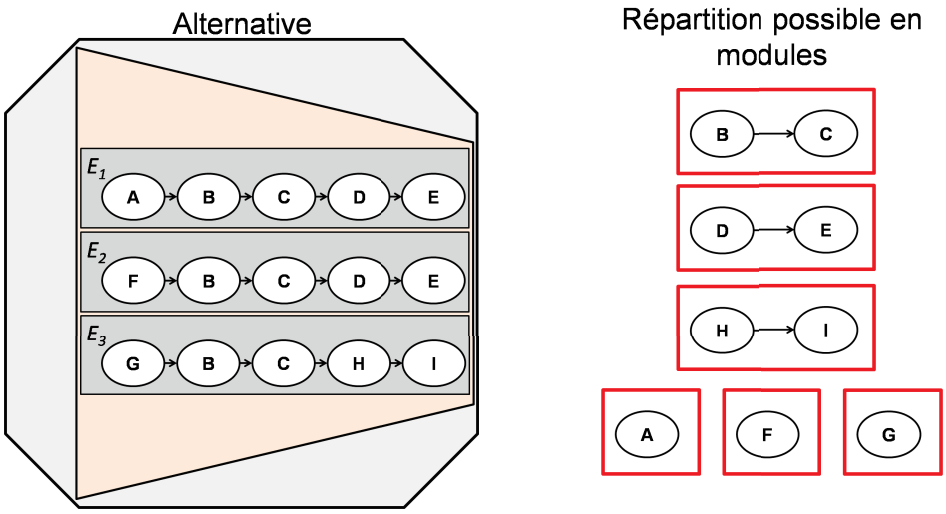

FIGURE 7.6 – Exemple de répartition des *Atomes* dans les modules au sein d'une *Alternative*

utilisés dans E_3 et seulement dans E_3 et peuvent être regroupés dans le même module. Enfin, les *Atomes* A, F et G ne peuvent se trouver que dans des modules distincts.

Le traitement du mécanisme de commutation dépend de la complexité de l'*Alternative*, c'est-à-dire de l'importance des changements à effectuer et du nombre de modules impactés. Nous distinguons trois niveaux de complexité :

- *Alternative locale* : traduit l'application de modifications très localisées sur la *Composition*. Dans la grande majorité des cas, toutes les *Entités Composables* de jonction de cette *Alternative* seront l'*Atome Vide* (i.e. pas d'actions spécifiques à exécuter à la commutation). Cette alternative ne va concerner qu'un module ou même que certaines entités utilisées dans ce module.
- *Alternative partielle* : traduit une modification d'une partie d'une fonctionnalité (activation d'une partie de la fonctionnalité après son initialisation, changement de la manière de réaliser une fonctionnalité). Elle va généralement concerner la commutation d'un petit nombre de modules appartenant au même schéma. Ces changements seront induits par l'état du système (par exemple une valeur d'un capteur).
- *Alternative globale* : traduit un remplacement d'une ou plusieurs fonctionnalités qui entraîne une restructuration d'une grande partie de la *Composition* (par exemple, passage de la téléopération du cap du robot à l'asservissement de ce même cap comme décrit à l'Exemple 5.8). Elle peut concerner un nombre important de modules et peut même affecter plusieurs schémas. Il s'agit en outre souvent de changements en partie contrôlables par l'utilisateur.

Règle 11:

Une *Alternative locale* sera mise en œuvre sous la forme d'une **instruction conditionnelle** portant sur l'*Entité Composable* de sélection qui sera de fait intégrée directement au module dans lequel est mise en œuvre l'*Alternative*.

Règle 12:

Une *Alternative partielle* sera mise en œuvre via un envoi de paramètres au module qui seront utilisés dans une **instruction conditionnelle** et les *Entités Composables* de jonction seront effectuées directement au sein du module dans lequel est mise en œuvre l'*Alternative*.

Règle 13:

Une *Alternative globale* sera mise en œuvre sous la forme d'un **changement de schémas**. Les *Entités Composables* de jonction pourront soit être activées directement dans le ou les modules contenant l'entité substituable via un paramètre fixé par le superviseur (cas où elles sont peu nombreuses), soit utilisées directement par le superviseur ou un module asynchrone manipulé par ce dernier puis le résultat sera transmis sous forme de paramètres aux modules concernés.

Il est à noter qu'originellement les *Alternatives partielles* devaient être implémentées de la même manière que les *Alternatives* globales. Néanmoins, comme cela a été souligné pour la Règle 7, nous avons parfois dû regrouper des modules dans des mêmes schémas afin d'avoir de meilleures garanties sur le déterminisme de notre application. De fait, les changements relativement localisés de ce type d'*Alternative* engendraient malgré tout une commutation de schémas sur une grande partie de l'application d'où un coût temporel important pour la commutation, coût incompatible avec des modifications de fonctionnalité demandées par le contrôleur et nécessitant, souvent, une réponse rapide pour préserver la réactivité du contrôle. Nous avons donc dû adopter une solution "dégradée" mais les évolutions envisagées du *Middleware* ContrACT, notamment sur l'exécution des schémas, nous permettront de résoudre ces problèmes et d'unifier les mécanismes d'implémentation des deux approches.

7.2.6 Liens

Ici, nous traitons naturellement des *Liens* qui sont effectués entre des entités situées dans des modules différents.

Règle 14:

Les *Liens* vers des *Besoins* avec des *Contraintes* de *nature Périodique* ou *Couplé* seront mis en œuvre sous forme de **flux de données**.

Ces *Liens* relient des *Atomes* s'exécutant de manière périodique, qui sont donc contenus dans des modules synchrones et liés entre eux par des flux de données.

Règle 15:

Les *Liens* entre un *Atome* dont la *Physique* a une *Contrainte* de *nature Sporadique* et un *Besoin* de *nature Sporadique* seront mis en œuvre différemment suivant le type de module destination et le type de module source. Les différents cas sont présentés dans le Tableau 7.1

Tableau 7.1 – Types de liaisons intermodules utilisées pour mettre en œuvre des *Liens* de *nature Sporadique*

Module source \ Module cible	Module synchrone	Module asynchrone
Module synchrone (Règle 2)	Flux de données	Flux d'évènements
Module asynchrone (Règle 3)	Envoi d'un évènement à un superviseur puis fixation de paramètre(s)	Flux d'évènements

Si les trois autres cas sont triviaux, le fait de créer un lien depuis un module asynchrone vers un module synchrone est plus intéressant. En effet, les modules asynchrones ne peuvent qu'envoyer des évènements. Par contre, de par leur définition dans ContrACT, les modules synchrones ne peuvent pas recevoir d'évènements. Dès lors, une connexion directe est impossible. De fait, la seule solution est de passer par un superviseur qui va servir de relais. Le module asynchrone va transmettre un évènement au superviseur. Ce dernier, en retour, va venir fixer un (ou plusieurs) paramètre(s) du module synchrone permettant ainsi de mettre à jour la donnée de manière sporadique. Cette procédure est schématisée à la Figure 7.7.

7.2.7 Paramètres temporels

Règle 16:

La **durée critique** d'un module synchrone sera déterminée par diffusion de la *Propriété* $t_{comp_{max}}$ des *Atomes* qu'elle contient selon la formule 6.1.

Règle 17:

La **période d'un schéma** sera déterminée par la valeur de T_{exe} choisie pour la partie de *Composition* qu'il implémente.

Règle 18:

Si un schéma peut avoir différentes périodes au cours de l'exécution (à cause d'un changement dans une *Alternative* par exemple) alors deux solutions sont possibles. Il est possible soit de réaliser un **changement de schéma** si l'on en possède un par période, soit de lui fixer une **période unique qui est le PGCD** [a] **des différentes périodes d'exécution possibles**. Dans ce cas, il ne faudra pas oublier **d'adapter le nombre de cycles séparant les envois de données** par ces modules en fonction de chaque période.

a. PGCD : Plus Grand Commun Diviseur

Pour expliquer ces choix, il est nécessaire de rappeler qu'il est actuellement impossible, sous ContrACT, de changer la période d'un schéma au cours de l'exécution. Donc un changement de période dynamique doit se traduire par un changement de schéma. Néanmoins, cette méthode oblige à dupliquer la définition des schémas et nécessite, de plus, une commutation temporellement coûteuse. La seconde option est donc de poser une période d'exécution unique à notre schéma. Cela permet de réduire les duplications et de supprimer la commutation mais impose une charge calculatoire supérieure à celle imposée par les périodes puisque les calculs s'effectueront à une période au moins aussi basse que la plus basse des périodes d'exécution.

Le choix de l'une ou l'autre des méthodes doit donc se baser sur les raisons ayant motivé le changement de période. Par exemple, si celui-ci est imposé afin d'alléger la charge calculatoire du processeur (pour mettre en œuvre, par exemple, des mécanismes issus de l'*autonomic computing*) alors la seconde option n'est clairement pas adaptée et la première devra forcément être choisie. Inversement, si par exemple, l'application nécessite des modifications fréquentes de périodes alors les nombreuses commutations deviennent coûteuses rendant de fait la seconde option plus pertinente dans ce cas.

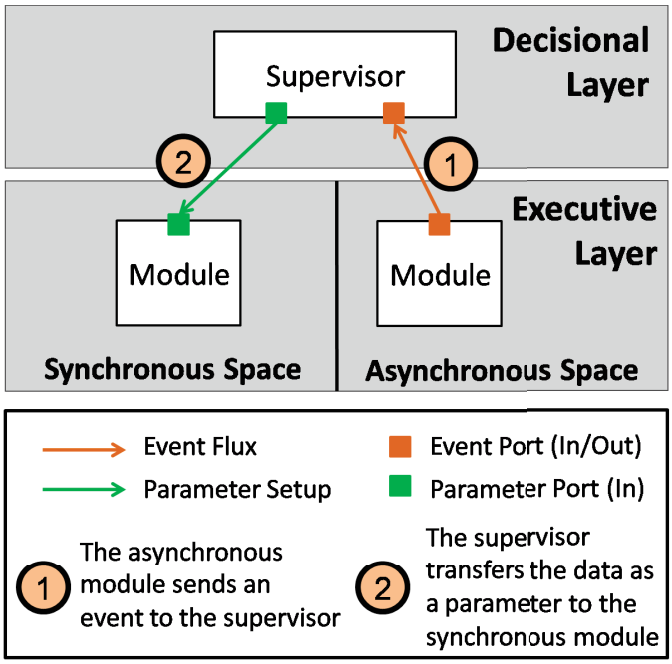

FIGURE 7.7 – Illustration de l'établissement d'une liaison entre un module asynchrone et un module synchrone

7.3 Points clés du chapitre

- La phase de projection, spécifique au *Middleware* cible, permet de guider le concepteur du logiciel de contrôle dans le choix des mécanismes d'implémentation des différentes entités et dans le paramétrage du logiciel.
- Cette phase se base sur un ensemble de règles "d'expérience" spécifiques au *Middleware* cible, le *Middleware* ContrACT en l'occurrence, utilisé pour développer nos logiciels de contrôle.
- Celui-ci s'organise autour de briques logicielles appelées modules. Ces derniers appartiennent soit à la couche décisionnelle (modules superviseurs chargés de gérer l'exécution des autres modules en fonction des objectifs de mission), soit à la couche exécutive (qui met en œuvre les différentes fonctionnalités robotiques).
- Les modules exécutifs sont soit asynchrones (d'exécution évènementielle) soit synchrones (d'exécution périodique).
- Les modules synchrones sont regroupés en schémas dont l'exécution est gérée par un module ordonnanceur. Chaque schéma possède sa propre période d'exécution.
- Pour permettre son ordonnancement, chaque module se voit associé une durée d'exécution.
- La phase de projection va nous permettre de déterminer la répartition de nos différentes entités entre les modules, les regroupements des modules périodiques entre les schémas, les périodes d'exécution de ces mêmes schémas et les durées d'exécution allouées à chaque module.

Quatrième partie

Exemples et expérimentations

Chapitre 8

Dispositif Expérimental et approche : de la simulation à l'expérimentation

Lors de la deuxième partie de ce manuscrit, nous avons présenté un formalisme permettant une description modulaire d'une architecture de contrôle. Lors de la partie suivante, nous avons décrit la méthodologie qui, s'appuyant sur ce formalisme, permet de guider son implémentation sur notre cible applicative. Il est maintenant temps d'illustrer notre approche sur divers exemples applicatifs concrets.

Nous allons dans un premier temps présenter le vecteur robotique qui servira de support à nos expérimentations. Mais, de par la complexité du développement d'une loi de commande pour une application robotique, et plus encore de par la complexité du milieu applicatif qui est notre objectif, l'environnement karstique, il n'est pas envisageable de passer directement à la phase d'expérimentation. Nous allons donc également introduire les différentes étapes qui permettent une validation incrémentale de l'architecture de contrôle depuis sa description jusqu'à sa mise en œuvre.

8.1 Le Jack et ses différentes versions

8.1.1 La version de base et ses limitations

A la section 1.3, nous avons présenté la version de base du Jack. Celle-ci est équipée d'une centrale inertielle et d'un capteur de profondeur. Ces capteurs permettent de réaliser des asservissements basiques tels que la tenue en cap ou en profondeur qui sont utilisés dans quasiment toutes les applications robotiques. Ils permettent notamment de simplifier le pilotage du robot. Néanmoins ces capteurs sont insuffisants pour les besoins des applications que nous souhaitons réaliser.

Nous allons donc utiliser d'autres capteurs. Il est ainsi nécessaire d'employer un Loch Doppler afin d'obtenir les vitesses du robot. Ce capteur est indispensable car l'intégration des accélérations fournies par l'IMU présente une dérive trop importante pour que les vitesses soient utilisables dans le cadre de nos applications, que ce soit pour les asservissements ou pour rendre la navigation plus précise et ainsi améliorer, par exemple, la reconstruction de l'environnement. Le modèle retenu est le *Sea*PILOT fabriqué par la société Rowe Technologies [Tec13].

Dans le cadre de l'exploration karstique, notre objectif, nous avons besoin de mettre en œuvre notre fonctionnalité d'évitement de parois et il nous faut également acquérir des mesures afin de reconstruire un modèle du réseau karstique. Pour répondre à ces besoins, nous avons choisi d'équiper le robot d'un sonar profilométrique qui permet de réaliser des mesures à 360 degrés sur un plan. Celui-ci est monté de telle manière qu'il réalise la prise d'un plan de coupe perpendiculaire à l'axe d'avancée du robot. Le capteur équipant notre robot est le Super SeaKing produit par la société Tritech [Tri15].

L'architecture matérielle du Jack dans sa version de base est décrite Figure 8.1. Les batteries et l'électronique de puissance ne sont pas représentées sur le schéma. Les capteurs présents de base sont connectés à la BeagleBone via un bus I^2C. Les variateurs qui commandent les moteurs sont également connectés sur ce bus. Un *switch* assure la connexion entre la BeagleBone, la caméra IP, le PC de supervision en surface (via l'ombilical) et les capteurs de charge utile.

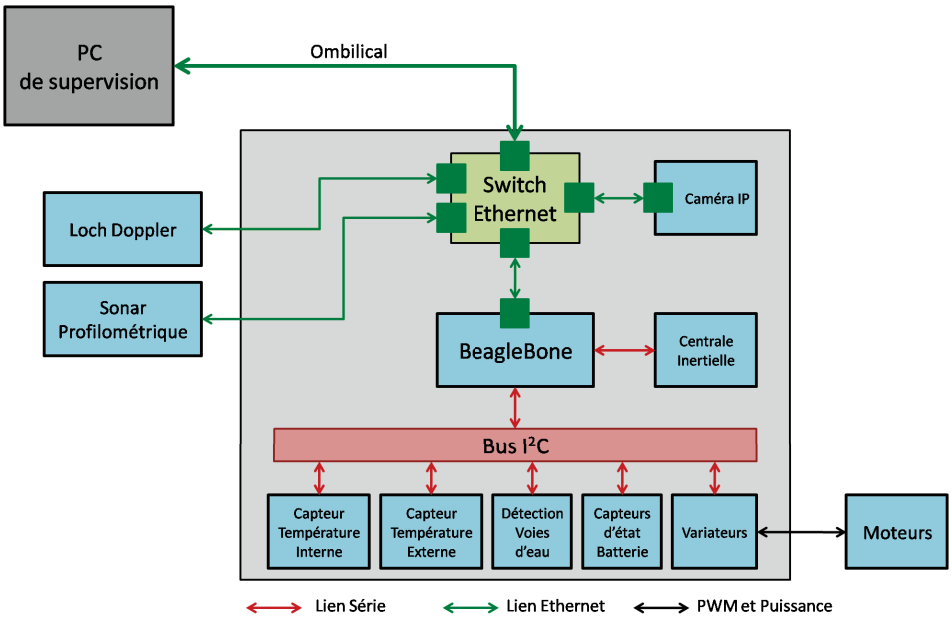

FIGURE 8.1 – Architecture Matérielle du ROV Jack dans sa version 6 moteurs

Néanmoins, si l'architecture matérielle basique permet d'ajouter facilement les capteurs

nécessaires, l'utilisation simultanée du Loch Doppler et du sonar profilométrique n'est pas directe. En effet, il s'agit tous deux de capteurs très volumineux qui doivent être montés sur un *skid* (voir Figure 1.5). Néanmoins, leur encombrement et leur masse importante pénalisent grandement le pilotage du robot car ils modifient son comportement hydrodynamique.

Considérons un mouvement d'avance du robot. Dans la version de base du Jack, le métacentre (barycentre entre le centre de gravité et le centre de flottabilité) du robot, la force de poussée générée par les actionneurs $\vec{F_u}$ et la trainée générée par le déplacement dans l'eau $\vec{F_{d_{body}}}$ sont coplanaires comme illustré Figure 8.2 (a). Dès lors, la dynamique de tangage n'est pas sollicitée.

Lorsque nous ajoutons le *skid*, nous perdons cette propriété de stabilité du tangage car deux phénomènes perturbateurs apparaissent. Tout d'abord, la masse importante des capteurs modifie la position du métacentre. Celui-ci n'est dès lors plus situé dans le plan d'actionnement horizontal ce qui entraîne l'apparition d'un couple $\vec{\Gamma_u}$ comme montré Figure 8.2 (b). Le second phénomène provient de la résistance hydrodynamique induite par l'imposant bloc formé par le *skid* et les capteurs. Cela crée une force de trainée $\vec{F_{d_{skid}}}$. Le déséquilibre de celle-ci avec $\vec{F_{d_{body}}}$ génère alors l'apparition d'un couple $\vec{\Gamma_d}$. Ces deux couples causent une rotation du tangage qui ne peut plus être compensée par l'actionnement du Jack dans sa version 6 moteurs.

Une option pour réduire ces phénomènes est de limiter drastiquement la vitesse d'avance u du robot. En effet, si l'on note u_{des} la vitesse d'avance souhaitée, on a donc :

$$\|\vec{F_u}\| = f(u_{des}) \tag{8.1}$$

$$\|\vec{F_{d_{body}}}\| = g(u) \tag{8.2}$$

$$\|\vec{F_{d_{skid}}}\| = h(u) \tag{8.3}$$

Diminuer les plages possibles de u et u_{des} réduit donc l'amplitude des différentes forces et permet ainsi de grandement limiter les couples induits. Néanmoins, cette option n'est evidemment pas viable dans la plupart des applications visées.

Pour résoudre ce problème, il faut donc actionner le tangage pour pouvoir le contrôler et l'asservir sur la valeur souhaitée. En outre, dans des environnements tels que les aquifères karstiques, les conduits sont loins d'être tous horizontaux et de fait, le besoin de maintenir le robot dans l'axe des conduits (ce qui améliore à la fois la qualité de la reconstruction et l'efficacité de l'évitement de parois) implique un actionnement du tangage. De plus, vu l'encombrement des différents capteurs, nous souhaitons pouvoir disposer d'une plus grande puissance d'actionnement afin de pouvoir les transporter malgré la traînée accrue qu'ils génèrent.

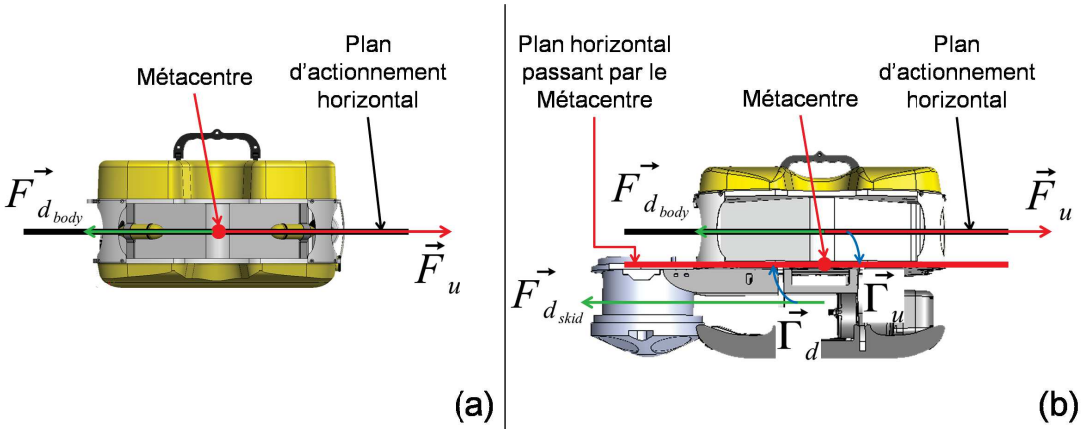

FIGURE 8.2 – Dans sa version de base, le tangage du Jack est naturellement stabilisé (a). L'ajout des capteurs et du skid fait perdre cette propriété (b)

8.1.2 Version douze moteurs

Tous ces aspects nous ont donc poussés à développer une autre version du ROV Jack avec douze moteurs. Celle-ci, montrée à la Figure 8.3, est constituée de deux étages d'actionnement du Jack reliés ensemble par un *skid*.

Dans la Figure 8.3, le robot est équipé à l'avant d'un sonar sectoriel (qui ne sera pas utilisé ici). Le sonar profilométrique est monté à l'arrière (il peut également être monté à l'avant en l'absence du sonar sectoriel). Enfin, le Loch Doppler est monté verticalement au centre du robot.

Dans cette version, l'étage du bas et celui du haut sont reliés ensemble par le bus I^2C. L'étage du bas ne contient en outre pas de contrôleur et est donc dépendant des consignes transmises par l'étage du haut. Il porte qui plus est sa propre alimentation en énergie. Un *switch* Ethernet sur l'étage du bas permet de connecter trois capteurs de charge utile dont les données sont remontées à l'étage du haut par une liaison Ethernet. L'architecture matérielle du robot est schématisée à la Figure 8.4.

De par ces modifications, le tangage devient contrôlable par différentiel entre les commandes appliquées aux actionneurs du haut et à ceux du bas. En outre, la puissance supplémentaire fournie par les moteurs ajoutés facilite les mouvements du robot puisque l'on dispose de plus de puissance propulsive. Ces modifications s'accompagnent bien entendu d'une augmentation sensible du volume et de la masse du robot. Cependant, avec des dimensions de 54 cm de long, 41 cm de large et 70 cm de haut et un poids maximal (i.e. tous capteurs charge utile compris) d'environ 50 kg, il demeure compatible avec les contraintes induites par les environnements considérés dans nos travaux.

Néanmoins, si les problématiques liées à l'intégration mécanique des capteurs sont résolues

FIGURE 8.3 – Le ROV Jack dans sa version 12 moteurs

avec cette version du Jack, il reste encore à traiter de la problématique de leur intégration logicielle. D'un côté le Loch Doppler transmet les données sous forme de trames NMEA envoyées toutes les 100 millisecondes (à sa période maximale de fonctionnement). Le volume de données transmis est donc suffisamment léger pour être traité en ligne par la BeagleBone. Ce n'est, par contre, pas le cas des données envoyées par le sonar profilométrique.

En effet à chaque *ping* (i.e. envoi d'une onde acoustique sur l'angle de mesure), toutes les 25 millisecondes environ, le sonar transmet une *scanline* découpée en plusieurs trames. La *scanline* représente l'intensité de l'écho sonar en fonction de la distance comme illustré Figure 8.5.

Une fois reconstituée à partir des trames, celle-ci doit encore être traitée pour identifier, pour chaque *ping*, la distance réelle de l'impact. Dès lors, ce traitement n'est pas réalisable en ligne sur la BeagleBone en plus des asservissements mis en œuvre, même en considérant que le *log* des données sonar s'effectue sur le PC de supervision.

Il faut alors trouver d'autres solutions. Une option possible est de déporter les calculs sur le PC de supervision. Néanmoins, celle-ci n'est réaliste que si les données que doit fournir

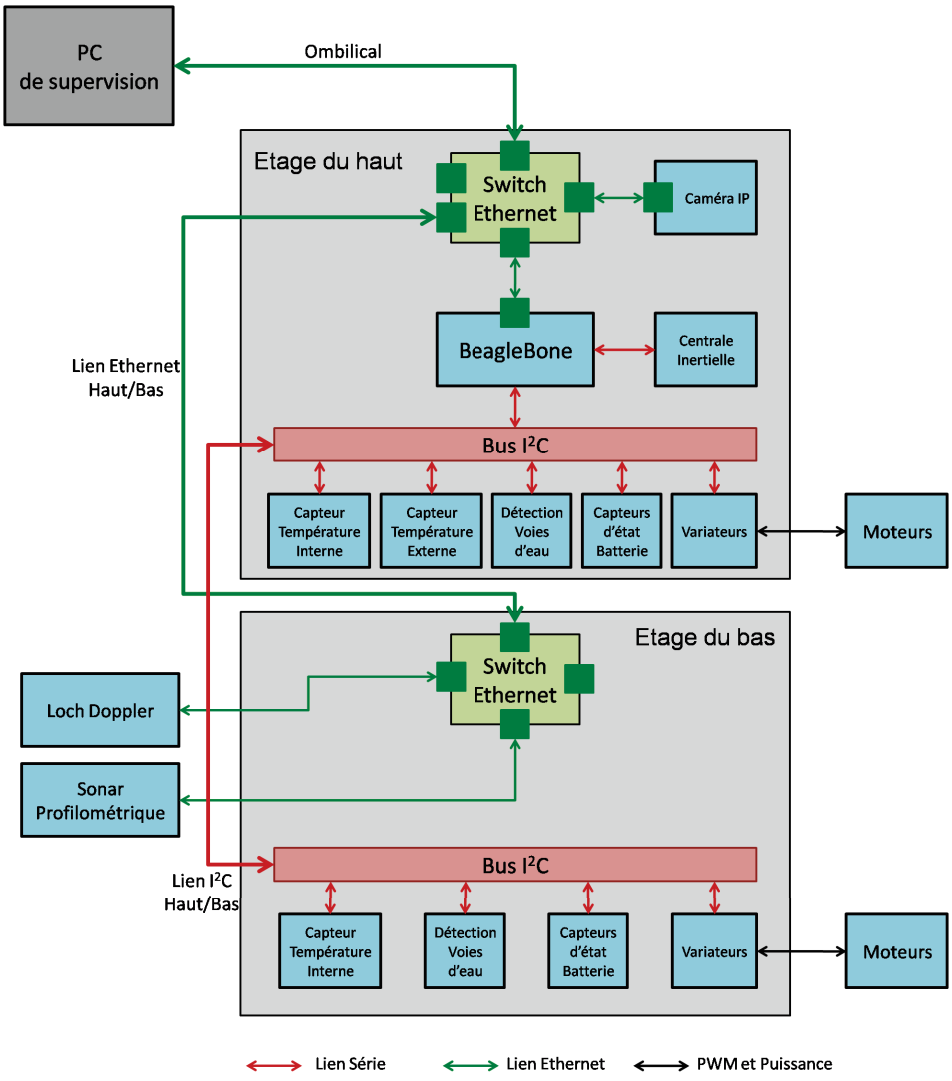

FIGURE 8.4 – Architecture Matérielle du ROV Jack dans sa version 12 moteurs

le PC de supervision sont transmises à une période suffisamment élevée pour que les délais induits par la communication (i.e. transferts via l'ombilical vers le PC de supervision puis du PC déporté à la BeagleBone) deviennent négligeables devant la période de mise à jour des données. En outre, cette option impose le maintien permanent d'une liaison entre le robot et le PC de supervision. Il ne devient alors plus possible de se séparer du câble si celui-ci devient problématique.

8.1.3 Ajout d'un contrôleur haut-niveau

De fait, pour pouvoir à la fois traiter un volume important de données mais également mettre en place des asservissements plus complexes nécessitant plus de puissance de calcul que celle disponible sur la BeagleBone, nous pouvons ajouter un contrôleur supplémentaire monté

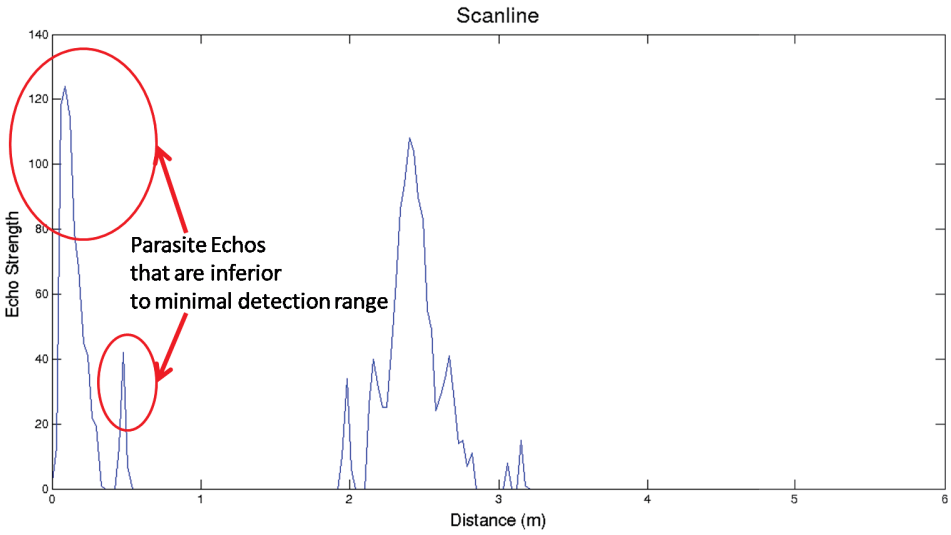

FIGURE 8.5 – Un exemple de *scanline*

sur le robot dans un caisson étanche indépendant. En outre, dans cette version, qui nécessite également une modification des pénétrateurs de coque pour pouvoir relier le contrôleur de haut-niveau au robot, nous avons remplacé l'ombilical cuivre par une liaison fibre optique qui offre de meilleurs débits tout en autorisant des longueurs de câble bien plus importantes.

L'architecture matérielle résultante est présentée Figure 8.6. Dans cette version, la BeagleBone continue de gérer l'électronique bas-niveau du robot et les asservissements basiques tandis que le calculateur de haut-niveau utilisera les services proposés par la BeagleBone afin de réaliser des asservissements plus avancés mettant en œuvre d'autres capteurs comme le sonar profilométrique. Pour plus de détails, le lecteur pourra se référer aux travaux de thèse de Benoit Ropars sur le développement d'une architecture logicielle orientée service (Benoit Ropars, *Un vecteur robotique polyvalent pour l'exploration sous-marine faible fond*, thèse soutenue le 16 décembre 2015 à Montpellier, LIRMM).

Pour le contrôleur de haut-niveau, notre choix s'est porté sur l'Arbor EmCORE-i2305 [Arb15]. Cette carte possède une mémoire vive (RAM) de 8GB et également 4GB de mémoire eMMC. En outre son processeur 4 cœurs, Intel Atom E3800 [Cor15], nous permet de mettre en œuvre des traitements lourds sans pénaliser la réactivité de notre contrôle. En effet, si le Middleware utilisé, ContrACT, n'autorise pas, pour l'instant, de fonctionnement multiprocesseur, certains calculs lourds et non temps-réel tels que le log de données ou le traitement des mesures du sonar profilométrique voire, si besoin est, des traitements d'image (qui peuvent également être effectués sur le chipset graphique intégré) peuvent être déportés sur les cœurs non utilisés par ContrACT afin d'alléger les calculs de l'application temps-réel.

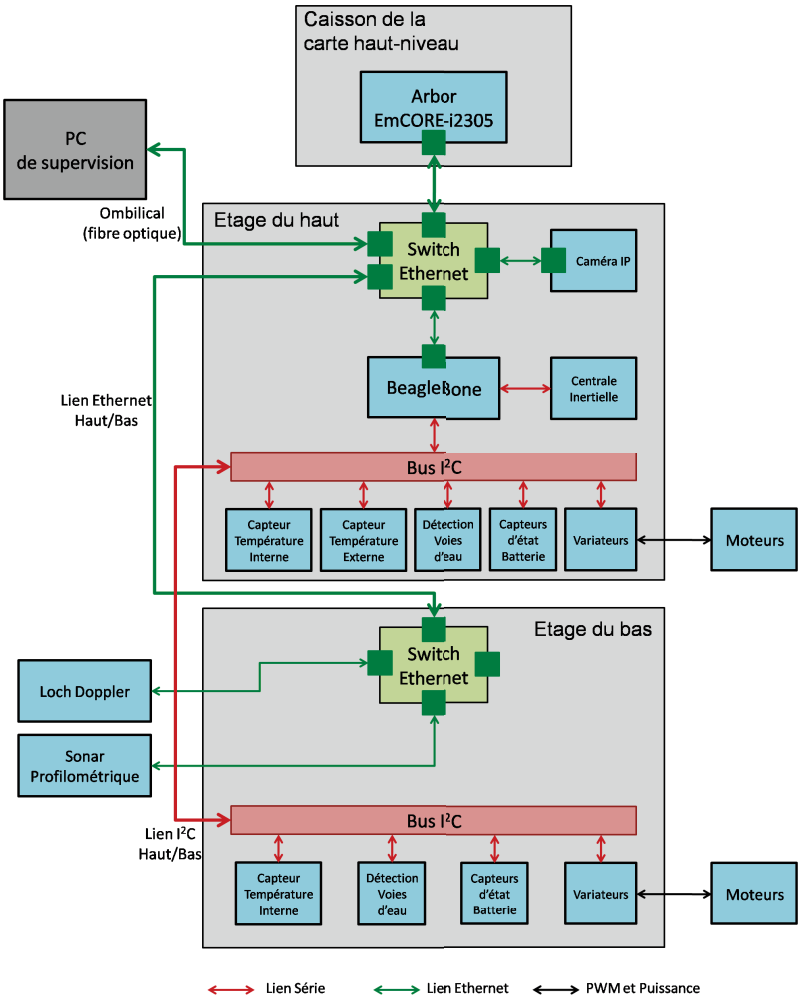

FIGURE 8.6 – Architecture Matérielle du ROV Jack dans sa version 12 moteurs, avec ajout d'un contrôleur de haut-niveau

8.2 Phases de simulation et d'expérimentation

Il est nécessaire de valider de manière progressive notre architecture de contrôle. Ainsi dans notre méthodologie, nous mettons en lumière 4 grandes étapes de validation. Accompagnant différentes étapes de la méthodologie, elles sont complémentaires les unes par rapport aux autres permettant une vérification incrémentale de l'architecture de contrôle et des lois de commande. Ces différentes étapes sont présentées à la Figure 8.7.

Dans nos travaux, les tests unitaires et d'intégration ne seront pas détaillés. Leur rôle est de permettre une vérification des entités logicielles implémentant nos *Atomes*.

Dans le cadre des tests unitaires, chaque entité est validée individuellement. Les *Atomes* encapsulant des connaissances indivisibles et minimales, cette étape se base sur les approches

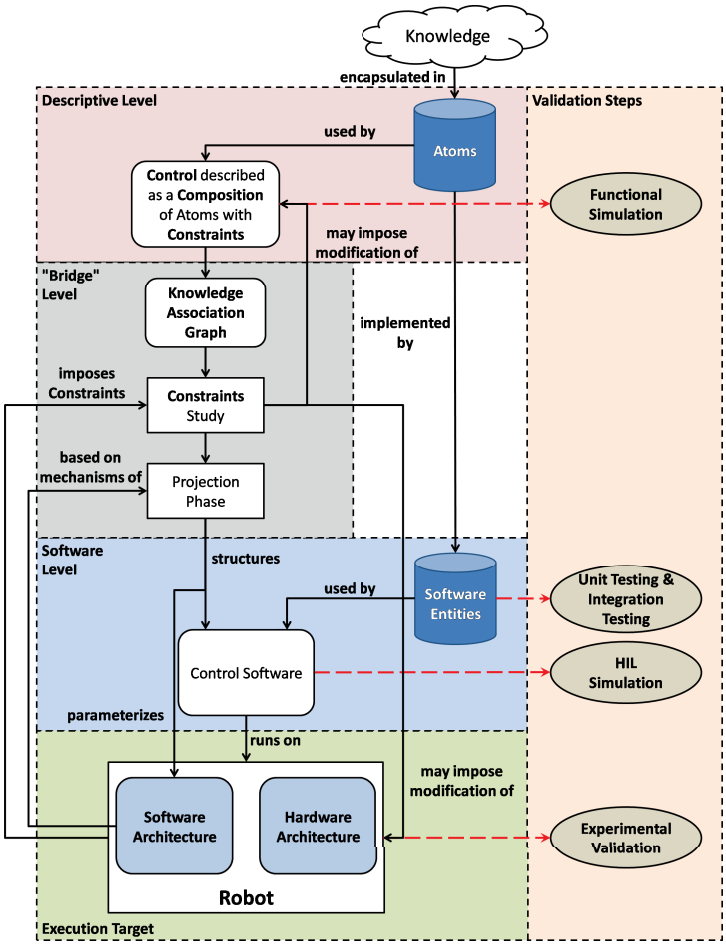

FIGURE 8.7 – Les grandes étapes de validation accompagnant notre méthodologie

utilisées par exemple dans le cadre de la validation de circuits intégrés. Ainsi le concepteur de l'*Atome* fournit en même temps que celui-ci un ensemble de vecteurs de tests à appliquer à ses *Besoins* et *Stockages Internes* ainsi que les valeurs attendues pour les *Produits* et les *Stockages Internes* après les calculs. Cela permet une vérification de la mise en œuvre de l'*Atome* et permet aussi, lorsqu'exécuté sur cible, de pouvoir déterminer le temps maximal de calcul de celui-ci.

Les tests d'intégration consistent quant à eux à valider le bon fonctionnement d'un groupe d'*Atomes* quand ils sont composés ensemble. Cette vérification peut également être basée sur des vecteurs de test. En outre, elle permet d'estimer les temps d'interconnexion (i.e. les temps mis par les différentes données échangées pour passer d'un *Atome* à un autre) bien que ceux-ci ne soient valables qu'en cas de connexion directe des *Atomes*. En effet, si ceux-ci n'appartiennent pas, par exemple, au même module ContrACT alors ces temps seront plus longs, puisqu'il faudra y ajouter les temps de conversion entre nos *Datatypes* et ceux de ContrACT ainsi que le temps de transfert des données d'un module à l'autre.

8.2.1 Simulation Fonctionnelle

La première étape consiste à valider les aspects fonctionnels des lois de commande décrites. Comme souligné dans la section 2.1.1, la simulation joue un grand rôle dans l'estimation des périodes d'exécution pour lesquelles notre contrôleur reste stable. Dès lors, la simulation fonctionnelle doit également permettre de faire varier les contraintes temporelles afin d'évaluer leur impact sur la stabilité du système.

Nous allons pour cela utiliser un "simulateur fonctionnel". Celui-ci est constitué de deux parties. D'un côté un programme Matlab gère le passage du temps, le modèle physique du robot, les capteurs et l'affichage. De l'autre, un programme C++ met en œuvre les entités logicielles implémentant les *Atomes* utilisés par la loi de commande et renvoie les consignes de commande à appliquer au robot. Les deux communiquent via un lien TCP qui assure une communication bloquante et donc une parfaite synchronisation temporelle entre les deux programmes. Il est en outre possible de modifier les périodes d'exécution d'une ou plusieurs entités logicielles afin de tester leur impact sur la stabilité.

De plus dans notre simulateur, l'étage d'actionnement n'est pas modélisé, de fait les consignes d'actionnement sont constituées du vecteur de forces exprimées dans le repère robot, F_B.

Le fonctionnement du simulateur est schématisé à la Figure 8.8.

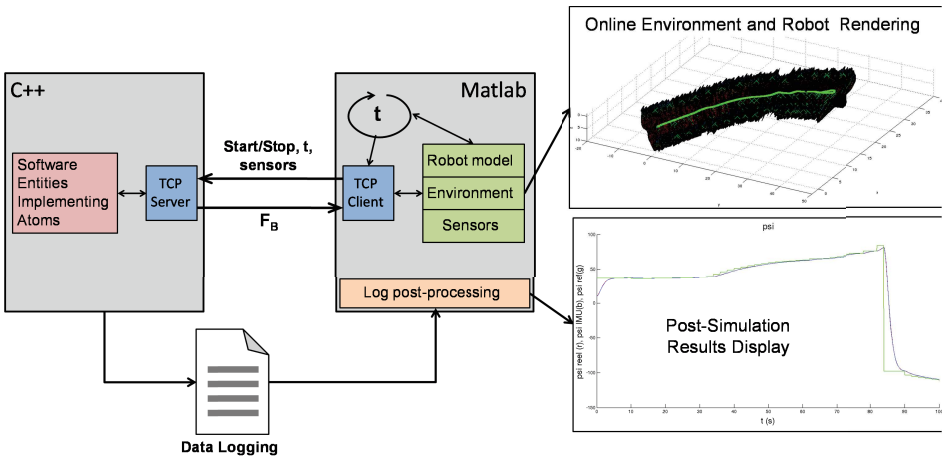

FIGURE 8.8 – Première étape : Simulation fonctionnelle

Néanmoins si la communication bloquante permet d'éliminer certains phénomènes tels que les désynchronisations qui complexifient la lecture des résultats de simulation, elle masque également l'impact de certains problèmes tels que les temps de calcul. Ainsi si le temps mis pour effectuer les calculs des *Atomes* dépasse la période fixée pour leur exécution, le problème ne sera pas apparent dans cette simulation.

Similairement l'étage d'actionnement n'a pas été modélisé car son impact peut perturber la lecture des résultats d'exécution de la loi de commande (saturations, zones mortes).

8.2.2 Simulation Hardware-in-the-Loop

La seconde étape permet de vérifier le logiciel de contrôle du robot via l'utilisation d'un simulateur *Hardware-in-the-Loop* (HIL).

Ce simulateur est constitué d'un programme Qt qui est chargé de simuler le robot, son environnement ainsi que les capteurs utilisés et son étage d'actionnement. Le simulateur est connecté au contrôleur bas-niveau du robot, la BeagleBone, par un lien UDP non bloquant. Sur celle-ci, le logiciel de contrôle est mis en œuvre à l'exception notable des *drivers* capteurs qui sont remplacés par des fonctions de communication avec le simulateur. Le fonctionnement est résumé à la Figure 8.9.

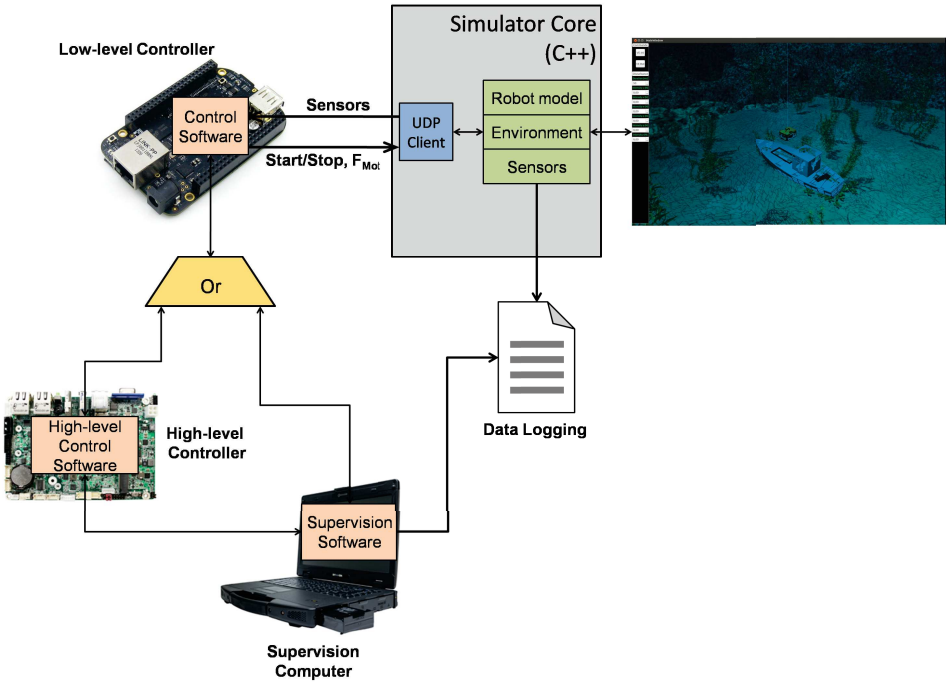

FIGURE 8.9 – Deuxième étape : Simulation Hardware-in-the-Loop

De fait ce simulateur va nous permettre tout d'abord de vérifier le bon fonctionnement du logiciel de contrôle notamment que l'ordonnancement s'effectue comme escompté, que les périodes d'exécution choisies pour les *Atomes* sont bien respectées et que les durées d'exécution des différents schémas respectent bien ces périodes. Cette simulation permet également une meilleure estimation de ces durées d'exécution en faisant apparaître des phénomènes tels que les préemptions qui sont difficiles à estimer de manière théorique et n'apparaissent pas lors d'une exécution individuelle de chaque *Atome*.

De fait si le fonctionnement du logiciel de contrôle n'est pas celui attendu, cette simulation va nous donner des pistes pour comprendre l'origine des problèmes rencontrés. En outre contrairement au simulateur fonctionnel, il est également possible de connecter à la BeagleBone la carte de haut-niveau ou le PC de supervision permettant ainsi de vérifier des fonctionnalités telles que la téléopération.

Cette simulation va nous permettre également de vérifier le bon fonctionnement du répartiteur qui est chargé de convertir les forces exprimées dans le repère robot en force à appliquer par les actionneurs. Le simulateur intègre donc un modèle des actionneurs.

Néanmoins les modèles utilisés, aussi précis soient-ils, ne sont jamais le reflet exact du robot réel. De plus le remplacement des *drivers* par une communication réseau ne permet pas d'évaluer leur impact à la fois en termes de charge calculatoire et de délais induits qui peuvent affecter le fonctionnement du logiciel de contrôle.

8.2.3 Expérimentations

Enfin, la dernière étape consiste à confronter le logiciel de contrôle au robot et à un environnement réel. Les différences entre la simulation et la réalité font qu'il est souvent préférable de valider dans un premier temps celui-ci dans un environnement contrôlé (i.e. une piscine) avant de se confronter aux environnements naturels.

Les expérimentations permettent donc de valider notre approche en confrontant le logiciel de contrôle au robot réel. De fait de nombreux points problématiques peuvent apparaître lors des tests. Les délais et la charge calculatoire induits par les *drivers* matériels du robot peuvent perturber le fonctionnement de notre architecture et, s'ils n'ont pas été convenablement estimés, remettre en cause les choix de périodes effectués obligeant à reprendre le processus de validation. De plus, les imprécisions de modélisation du robot imposent souvent de réajuster les coefficients de la commande afin de préserver sa stabilité. Enfin, des différences même faibles entre les performances des actionneurs influent sur le comportement du robot et ne peuvent être pleinement prises en compte qu'au stade de l'expérimentation.

8.3 Points clés du chapitre

- ► Nous allons utiliser un robot polyvalent qui peut être mis en œuvre en différentes versions suivant les besoins applicatifs.
- ► La validation d'une loi de commande jusqu'à son expérimentation sur un robot réel est un processus incrémental.
- ► La simulation fonctionnelle permet de vérifier le fonctionnement de la loi de commande et la validité des choix de périodes effectués.
- ► La simulation *Hardware-in-the-Loop* permet de vérifier le fonctionnement du logiciel de contrôle.
- ► Les expérimentations confrontent celui-ci au robot réel et à des problématiques difficilement modélisables en simulation.

Chapitre 9

Apport de la méthodologie et de la structuration à travers l'exemple d'un asservissement en cap

Nous allons maintenant souligner les apports de notre approche en nous basant sur l'exemple simple d'un asservissement en cap. Nous commencerons par présenter la fonctionnalité considérée. Nous illustrerons ensuite via des exemples de simulation HIL les problèmes, notamment en termes de stabilité, que soulève le non respect des *Contraintes* fixées. Cela nous permettra de souligner à nouveau l'importance du respect de ces contraintes temporelles. Enfin, nous montrerons, au travers du problème de la variation des caractéristiques des différents moteurs du robot, comment la structuration à base d'*Atomes* nous permet de mieux identifier les entités à faire évoluer pour résoudre un problème de contrôle permettant ainsi d'effectuer des modifications ciblées sans affecter les autres *Entités Composables*.

9.1 Présentation de l'exemple

Notre exemple va décrire l'implémentation d'un asservissement en cap. Nous conservons en outre notre volonté d'offrir des fonctionnalités simplifiant le pilotage du robot tout en laissant les choix décisionnels à l'opérateur humain. Ainsi, l'opérateur humain décidera du démarrage ou de l'arrêt de l'asservissement et choisira également la référence de l'asservissement en cap. Pour cela, lors du lancement de l'asservissement, la référence en cap sera alignée sur le cap actuel du robot (afin d'éviter une trop brusque variation du cap du robot) et l'opérateur pourra alors incrémenter ou décrémenter cette valeur pour obtenir la valeur désirée. Les autres degrés de liberté seront également commandés par l'opérateur (sans asservissement pour l'instant). L'application choisie est ainsi résumée Figure 9.1.

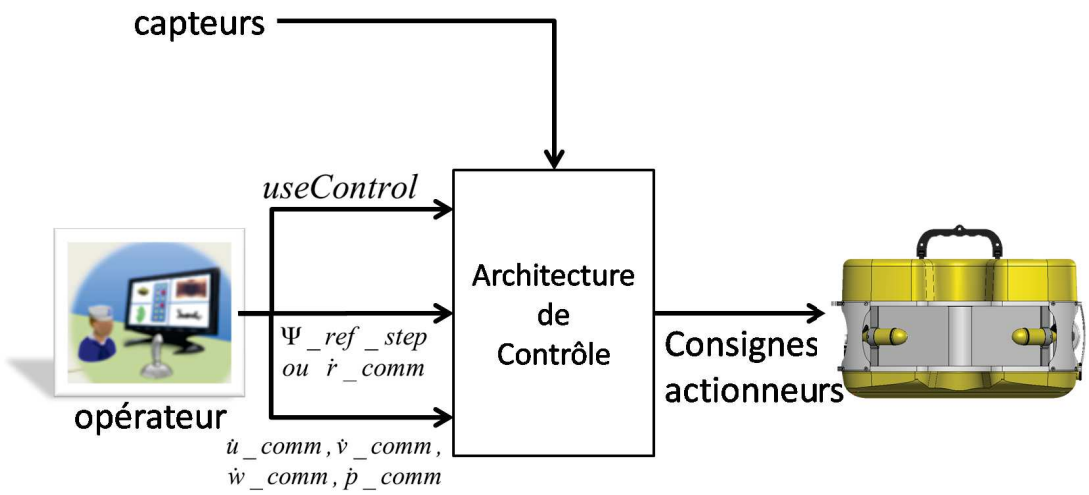

FIGURE 9.1 – Représentation schématique de la fonctionnalité d'asservissement en cap

Dans cette figure, *useControl* est le signal permettant d'activer l'asservissement en cap. Si celui-ci est actif, l'opérateur va transmettre les incréments à appliquer à la consigne en cap (Ψ_ref_step). Sinon, la rotation du robot sera directement commandée en accélération ($\dot{r}\_comm$). Similairement, les quatre autres degrés de liberté sont commandés en accélération et nous rappelons que dans la version de base du Jack, le tangage n'est pas actionné. Notre architecture de contrôle va déterminer les consignes à appliquer aux actionneurs. Enfin, les capteurs présentés dans le chapitre précédent seront utilisés pour la navigation du robot.

La description présentée ici se base sur l'Exemple 7.2 mais nous l'avons complétée afin qu'elle décrive l'exemple applicatif réel. Le *Graphe d'Association de Connaissances* décrivant la *Composition* est présenté Figure 9.2.

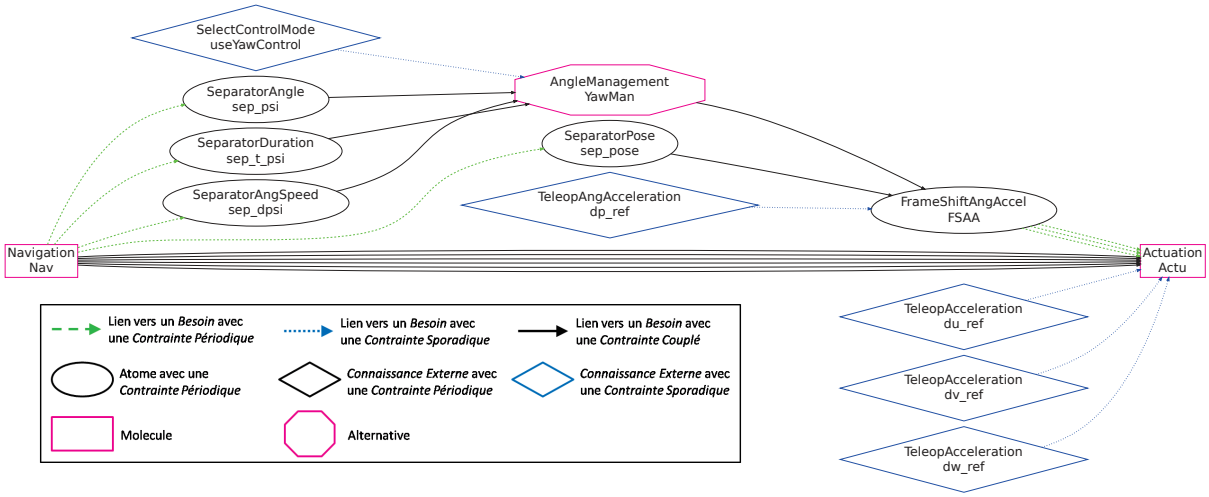

FIGURE 9.2 – *Graphe d'Association de Connaissances* décrivant cette fonctionnalité

9.1.1 Asservissement

L'*Alternative AngleManagement* est chargée de gérer la commutation entre le pilotage du cap via l'asservissement ou via une téléopération directe. Son fonctionnement est présenté en détails dans l'Exemple B.8 et nous ne intéresserons ici qu'à présenter plus précisément la *Molécule AngleControl* qui décrit l'asservissement en cap. Le *Graphe Moléculaire* associé à celle-ci est représenté Figure 9.3.

Considérons tout d'abord l'*Atome AnglePIDAngAccel*, qui est chargé d'implémenter le correcteur PID en lui-même. Ses *Paramètres d'Interface* sont :

$$IntPar(APAA) = \{Ent \in Fr, ax \in Ax\} \tag{9.1}$$

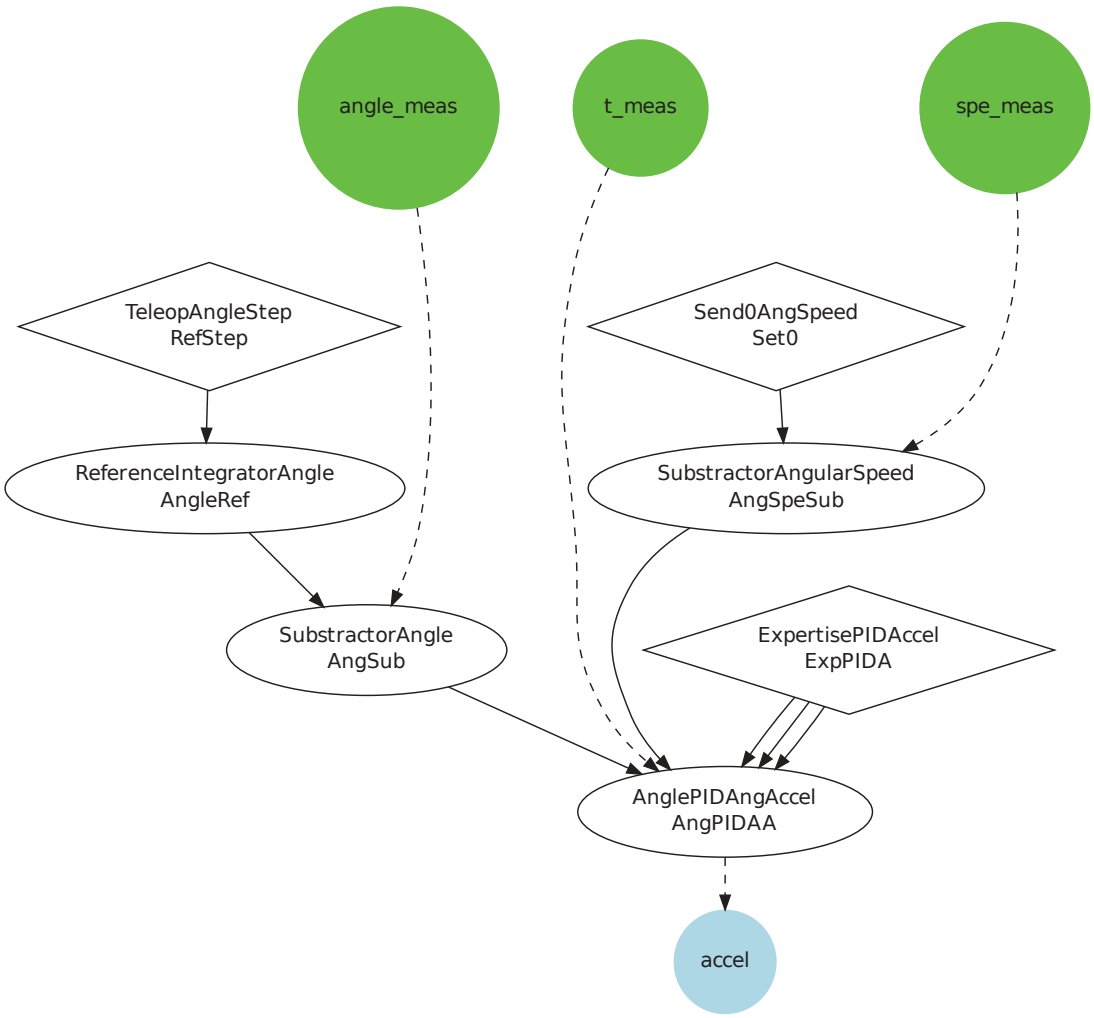

FIGURE 9.3 – La *Molécule AngleControl*

Nous pouvons ainsi définir $Int(APAA) = (Ne(APAA), Pr(APAA), IS(APAA))$ comme :

$$
\begin{aligned}
IS(APAA) &= Angle < Ent, ax > err\_tt && (9.2)\\
&\quad Boolean < NOFRAME, NOAXIS > init && (9.3)\\
&\quad Duration < NOFRAME, NOAXIS > t\_prec && (9.4)\\
Ne(APAA) &= Angle < Ent, ax > err && (9.5)\\
&\quad AngularSpeed < Ent, ax > derror && (9.6)\\
&\quad Duration < NOFRAME, NOAXIS > t\_err && (9.7)\\
&\quad FrequencySquared < NOFRAME, NOAXIS > Kd && (9.8)\\
&\quad Frequency < NOFRAME, NOAXIS > Kp && (9.9)\\
&\quad FrequencyCubed < NOFRAME, NOAXIS > KI && (9.10)\\
Pr(APAA) &= AngularAcceleration < Ent, ax > accel && (9.11)
\end{aligned}
$$

Et sa *Physique* est définie comme :

$$
\begin{aligned}
&if(init == true)\\
&\{\\
&\quad dt = t\_err - t\_prec;\\
&\quad err\_tt = err\_tt + error * dt;\\
&\}\\
&else\\
&\{\\
&\quad init = true;\\
&\}\\
&t\_prec = t\_err;\\
&accel = Kp * error + Kd * derror + KI * err\_tt;
\end{aligned}
$$

Lors de la conception de l'*Atome*, nous avons choisi de réaliser le calcul du terme intégral au sein de celui-ci car nous n'avons pas de capteur permettant de nous fournir cette information. La dérivée de l'erreur est quant à elle calculée séparément car nous disposons d'un capteur (la vitesse angulaire fournie par l'IMU) qui se révèle donc plus efficace qu'une dérivation numérique bien qu'en l'absence de capteur, nous puissions toujours recourir à cette solution. L'*Atome* $ExpertisePIDAccel$ est chargé de valuer Kp, KI et Kd. L'erreur angulaire est

calculée par l'*Atome SubstractorAngle* qui calcule la différence entre la consigne de cap et sa mesure actuelle. Il a pour *Interface* :

$$
\begin{aligned}
IS(SubAng) &= \emptyset && (9.12)\\
Ne(SubAng) &= Angle < Ent, ax > setpoint && (9.13)\\
&\quad\ Angle < Ent, ax > measure && (9.14)\\
Pr(SubAng) &= Angle < Ent, ax > error && (9.15)
\end{aligned}
$$

Sa *Physique* est définie comme :

$$
\begin{aligned}
&error = setpoint - measure;\\
&while(error > PI)\\
&\{\\
&\quad error- = 2 * PI;\\
&\}\\
&while(error < -PI)\\
&\{\\
&\quad error+ = 2 * PI;\\
&\}
\end{aligned}
$$

En plus d'effectuer la soustraction, cet *Atome* recale l'erreur entre $-PI$ et PI. Similairement, l'*Atome SubstractorAngularSpeed* calcule la dérivée de l'erreur à partir de la mesure capteur tandis que l'autre entrée de l'*Atome* est fixée à 0 par l'*Atome Send*0*AngSpeed*. Enfin, la référence en cap est calculée par l'*Atome ReferenceIntegratorAngle*. Il a pour *Interface* :

$$
\begin{aligned}
IS(SubAng) &= Angle < Ent, ax > ref\_pre && (9.16)\\
Ne(SubAng) &= Angle < Ent, ax > step && (9.17)\\
Pr(SubAng) &= Angle < Ent, ax > ref && (9.18)
\end{aligned}
$$

Sa *Physique* est :

$$
\begin{aligned}
&ref = ref\_pre + step;\\
&ref\_pre = ref;
\end{aligned}
$$

En sortie de l'*Alternative*, l'accélération angulaire désirée est exprimée dans le repère monde. L'*Atome* $FrameShiftAngAccel$ est donc chargé d'effectuer le changement de repère pour obtenir les accélérations angulaires exprimées dans le repère robot. Il utilise pour cela la position et les orientations du robot dans le repère monde sous forme d'un *Type* complexe, $CartesianPose$.

9.1.2 Navigation

Présentons maintenant la *Molécule* chargée d'effectuer la navigation du robot. Elle a pour *Produits* : les accélérations linéaires dans le repère robot ($\dot{u}$, $\dot{v}$, $\dot{w}$) mesurées par l'IMU, les vitesses linéaires exprimées dans le repère robot (u, v, w) mesurées par le Loch Doppler, les vitesses linéaires exprimées dans le repère monde ($\dot{x}$, $\dot{y}$, $\dot{z}$) qui sont calculées via le modèle cinématique à partir des vitesses du repère robot. $\dot{x}$ et $\dot{y}$ sont ensuite intégrées pour obtenir la position du robot dans le repère monde (x et y) et sa profondeur z est mesurée par le capteur de profondeur.

Les vitesses angulaires exprimées dans le repère robot (p, q, r) sont mesurées par l'IMU tandis que les vitesses angulaires exprimées dans le repère monde ($\dot{\phi}$, $\dot{\theta}$, $\dot{\psi}$) sont obtenues à partir du modèle cinématique du robot et l'orientation du robot dans le repère monde (ϕ, θ, ψ) est obtenue là encore par l'IMU. Les positions et orientations sont également regroupées dans un *Type* complexe, $CartesianPose$, simplifiant leur utilisation.

Les dates de ces différentes données font également partie des *Produits* pour être utilisées, si nécessaire, pour des intégrations ou pour le log de données. L'ensemble des *Produits* est résumé Figure 9.4.

Entre la *Molécule* de navigation et l'asservissement (*Alternative* $AngleManagement$ et *Atome* effectuant le changement de repère) nous avons placé des *Atomes Séparateurs*. En effet, les *Besoins* de l'*Alternative* (mesure de l'angle, de la vitesse angulaire et de la date de la mesure) et la pose du robot utilisée par l'*Atome* de changement de repère ont tous des *Contraintes* temporelles de *nature Couplé* puisque ces données doivent être mises à jour à chaque cycle d'exécution. Cela devrait donc entraîner un couplage temporel entre ces entités et la *Molécule* $Navigation$. Toutefois, ce couplage n'est pas souhaitable, notamment car, la navigation contenant des intégrations, il est souhaitable de la faire s'exécuter à la période la plus élevée possible.

Dès lors, les *Atomes* de la famille des *Séparateurs* permettent le découplage temporel au niveau d'un *Lien*. Ainsi, les *Contraintes* temporelles sont satisfaites mais, en même temps, les deux entités peuvent, si possible, fonctionner à des périodes différentes comme illustré Figure 9.5.

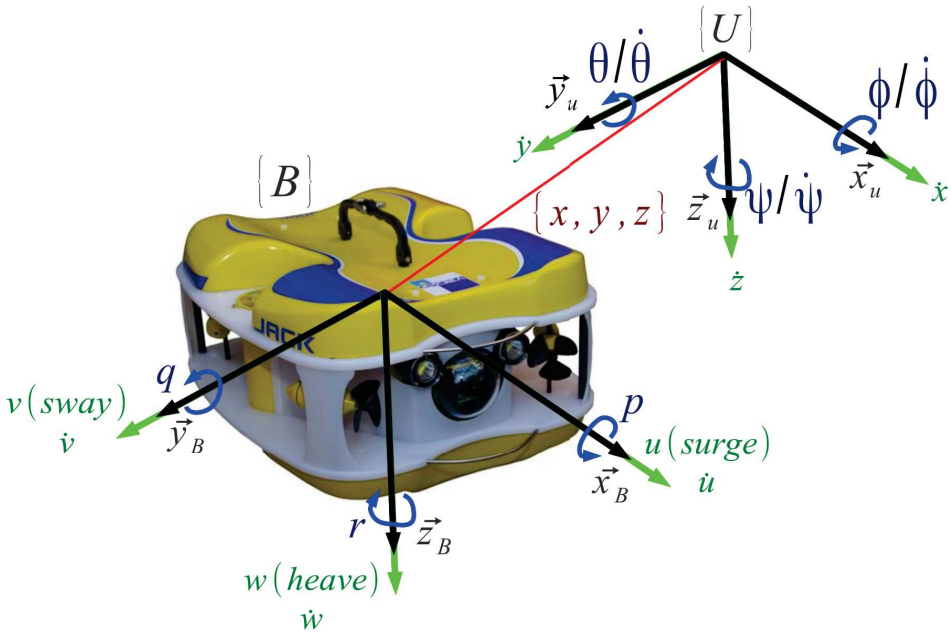

FIGURE 9.4 – Représentation spatiale des différents *Produits* de la *Molécule Navigation*

Par contre, nous n'avons pas découplé temporellement actionnement (qui est lui-même découplé temporellement de l'asservissement par le modèle dynamique) et navigation car, notamment à cause du modèle dynamique, l'actionnement doit lui aussi pouvoir fonctionner à la période d'exécution la plus faible possible.

9.1.3 Actionnement

Ici, la *Molécule Actuation* comprend à la fois le modèle dynamique (complet et non pas la version simplifiée présentée au Chapitre 4), le répartiteur, les caractéristiques moteur et enfin la *Connaissance Externe* qui représente les *drivers* des actionneurs.

La première partie de cette *Molécule* qui contient ses *Besoins* et les entités liées au modèle dynamique est représentée Figure 9.6.

Soit A un des six degrés de liberté du robot (u, v, w, p, q ou r), le modèle dynamique retenu (qui néglige les termes de couplage) est décrit comme :

$$F\_A = mA * dA + d\_A * A \tag{9.19}$$

où l'amortissement d_A est lui-même un terme dépendant de la vitesse sous la forme :

$$d\_A = -d\_stat\_A - d\_speed\_A * |A| \tag{9.20}$$

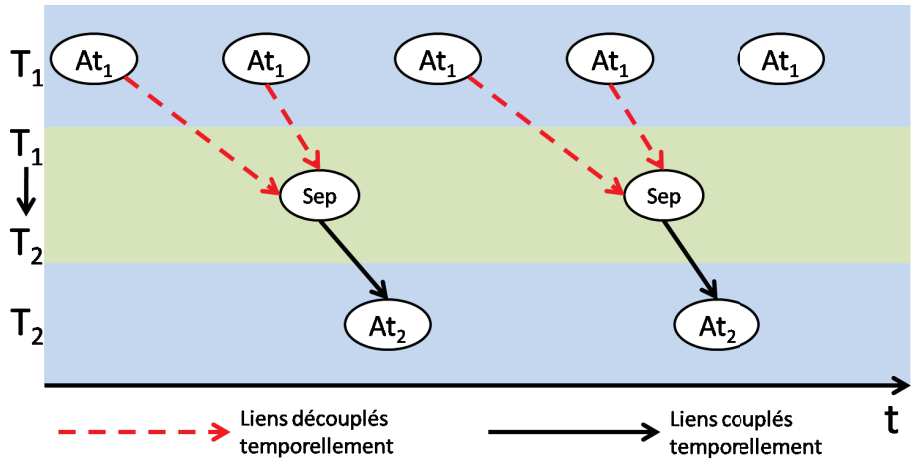

FIGURE 9.5 – L'*Atome Séparateur* permet à $A1$ et $A2$ de fonctionner à des périodes différentes tout en assurant leur découplage temporel

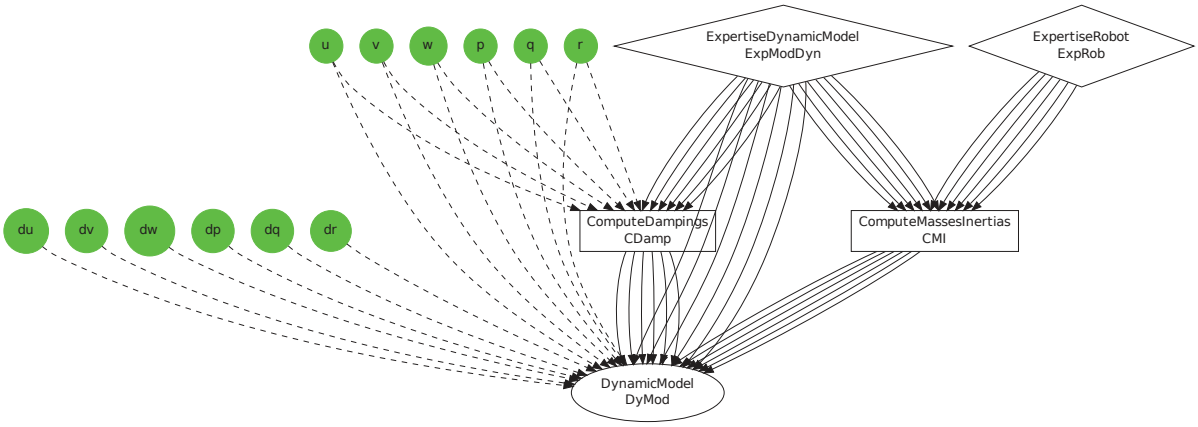

FIGURE 9.6 – *Graphe Moléculaire* de la *Molécule Actuation* (première partie)

Enfin, les masses (ou inerties) totales m_A sont calculées par :

$$mu = m - am\_u \tag{9.21}$$

$$mv = m - am\_v \tag{9.22}$$

$$mw = m - am\_w \tag{9.23}$$

$$mp = Ixx - aI\_p \tag{9.24}$$

$$mq = Iyy - aI\_q \tag{9.25}$$

$$mr = Izz - aI\_r \tag{9.26}$$

où m est la masse à sec (hors de l'eau) du robot et Ixx, Iyy, Izz les inerties suivant les

différents axes de rotation tandis que les autres termes désignent les masses et inerties ajoutées (pour prendre en compte l'effet de l'eau). Dans la Figure 9.6, l'expertise *ExpertiseRobot* permet de fixer à l'initialisation m, Ixx, Iyy et Izz tandis qu'*ExpertiseDynamicModel* value les masses et inerties ajoutées ainsi que les termes d'amortissement d_stat_A et d_speed_A.

Les équations (9.21) à (9.26) sont implémentées chacune dans un *Atome* dédié dans la *Molécule ComputeMassesInertias*. Les six équations de la forme (9.20) sont implémentées dans des *Atomes* spécifiques dans *ComputeDampings*. Enfin les équations de la forme (9.19) sont contenues dans la *Physique* de l'*Atome DynamicModel*.

La seconde partie de la *Molécule Actuation* est présentée Figure 9.7.

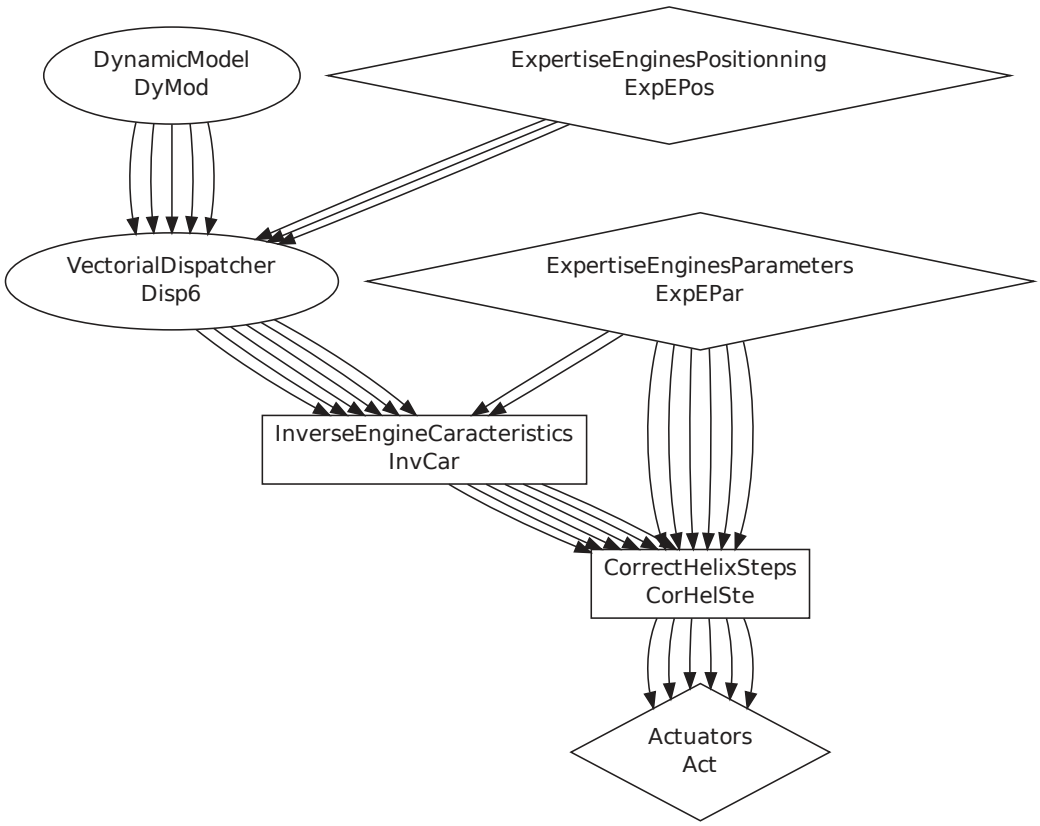

FIGURE 9.7 – *Graphe Moléculaire* de la *Molécule Actuation* (deuxième partie)

Tout d'abord, le modèle dynamique transmet les forces à appliquer dans le repère robot à l'*Atome VectorialDispatcher*. Celui-ci est le répartiteur géométrique présenté dans [RLLA15]. Les paramètres de positionnement des moteurs sont fixés par la *Connaissance Externe ExpertiseEnginesPositionning*. Il n'y a ici que cinq *Liens* entre ces deux entités puisque ce répartiteur ne peut pas contrôler le tangage (Fq).

Ensuite les caractéristiques[1] inverses des moteurs sont utilisées afin de déterminer le PWM[2] à appliquer à chaque moteur pour obtenir la force de poussée souhaitée. Cela est réalisé par la *MoléculeInverseEngineCaracteristics* qui contient six *Atomes* (un par moteur) *InverseLinearEngineCaracteristic*. Celui-ci suppose que les moteurs ont, sur la globalité de leur plage de fonctionnement, une caractéristique linéaire. Sa *Physique* est ainsi définie comme :

$$PWM = \frac{Fp - b}{a} \tag{9.27}$$

Dans un premier temps, nous supposerons que les coefficients des caractéristiques moteurs (a et b) sont identiques pour tous les moteurs. Nous avons estimé par mesure expérimentale sur les moteurs que $a = 0.31\ N$ et $b = 3.62\ N$ (pour assurer la cohérence des unités, les coefficients sont assimilés à des forces). Nous ne considèrerons pas pour l'instant le problème des zones mortes[3] [RLLA15]. Les valeurs de a et b sont fixées de manière statique à l'aide de la *Connaissance Externe ExpertiseEnginesParameters*.

Nous devons ensuite prendre en compte le pas[4] des hélices moteurs. Il est nécessaire de combiner les deux types de pas (à gauche et à droite) afin que les couples engendrés par la rotation des hélices se compensent et ne provoquent pas un couple de rotation qui ferait prendre naturellement du roulis au robot. Cela est réalisé par la *Molécule CorrectHelixSteps* qui comprend six *Atomes* indépendants (un par moteur) *CorrectHelixStep* chargés d'appliquer la correction de pas (1 pour une rotation de l'hélice vers la droite et -1 pour une rotation vers la gauche), avec pour *Physique* :

$$PWM\_out = PWM\_in * step \tag{9.28}$$

Les pas de chaque hélice sont valués à l'initialisation par "l'expert" humain via l'*Atome ExpertiseEnginesParameters*. Enfin, les consignes moteurs sont appliquées aux actionneurs via leur driver représenté par la *Connaissance Externe Actuators*.

1. Nos moteurs étant commandés en PWM, la caractéristique est définie comme la relation $Fp = f(PWM)$, où Fp est la force de poussée du propulseur.

2. Pulse Width Modulation, l'information est transmise par le rapport cyclique (i.e. le ratio entre la durée à l'état haut d'un signal par rapport à sa période) d'un signal périodique constitué de 0 et de 1.

3. Zones où l'application d'une consigne ne génère pas de poussée, i.e. $Fp = 0$, souvent car la force demandée est trop faible pour entraîner une activation du moteur ou pour que l'hélice parvienne à vaincre la résistance de l'eau et se mette à tourner.

4. Le pas d'une hélice correspond au sens de rotation qui lui fera produire une poussée de signe positif.

9.2 Simulation HIL et respect des contraintes

Nous allons dans un premier temps illustrer les conséquences possibles d'un non respect des *Contraintes* temporelles fixées via des résultats de simulation *Hardware-in-the-Loop*. Il s'agit là d'une version plus détaillée de l'exemple proposé dans [LRP+15]. Il nous faut préciser que dans [LRP+15] notre *Atome PID* avait été remplacé par un autre *Atome* qui réalisait un contrôle de type *PD* avec les coefficients choisis. Cela souligne l'évolutivité et la modularité de notre approche. Nous avons par contre choisi, dans cet exemple, d'illustrer l'exemple de l'*Atome PID*, à la fois car le terme intégrateur était nécessaire du point de vue expérimental et car il permettait de mieux mettre en évidence certains aspects de conception liés aux *Alternatives* notamment au niveau des *Entités Composables* de jonction.

Comme le simulateur ne nous impose pas de contraintes temporelles au niveau des capteurs (puisqu'il s'agit de capteurs virtuels), nous avons choisi de faire fonctionner les deux parties temporellement indépendantes à des périodes différentes. En effet, lors des expérimentations de la prochaine section et du chapitre suivant, les contraintes de fonctionnement de nos capteurs matériels (IMU, Loch Doppler et capteur de profondeur) ne nous permettront pas de faire cela et toutes les entités de la *Composition* devront fonctionner à une période identique même si elles ne sont pas temporellement couplées.

Nous avons ainsi choisi de faire fonctionner les *Atomes* des *Molécules Navigation* et *Actuation* à une période de 25ms. Les autres entités (*Alternative* contenant l'asservissement et changement de repère) fonctionneront à une période de 50ms.

9.2.1 Simulation Hardware-in-the-Loop, non respect des contraintes

La Figure 9.8 présente une de nos premières simulations de l'asservissement en cap effectuée avec le simulateur HIL. Dans cette simulation, notre PID avait pour paramètres $Kp = 10$, $Kd = 10$ et $KI = 0$. Comme nous pouvons le constater, après s'être stabilisé à une valeur proche du cap désiré, le comportement du robot devient instable et il commence à osciller avec de plus en plus d'amplitude. Il effectue ensuite deux tours sur lui-même avant de se stabliser à nouveau à sa valeur de consigne.

La cause de cette instabilité se trouve dans un non respect temporaire des *Contraintes* temporelles que nous nous sommes fixées.

La Figure 9.9 illustre l'occurrence du phénomène sur un des modules ContrACT utilisés (celui qui contient les *Atomes* effectuant l'asservissement en cap). Comme nous pouvons le constater, durant une grande partie de l'exécution, l'écart oscille autour de la période nominale du module (50ms). Mais, à trois reprises, la durée entre deux cycles successifs viole

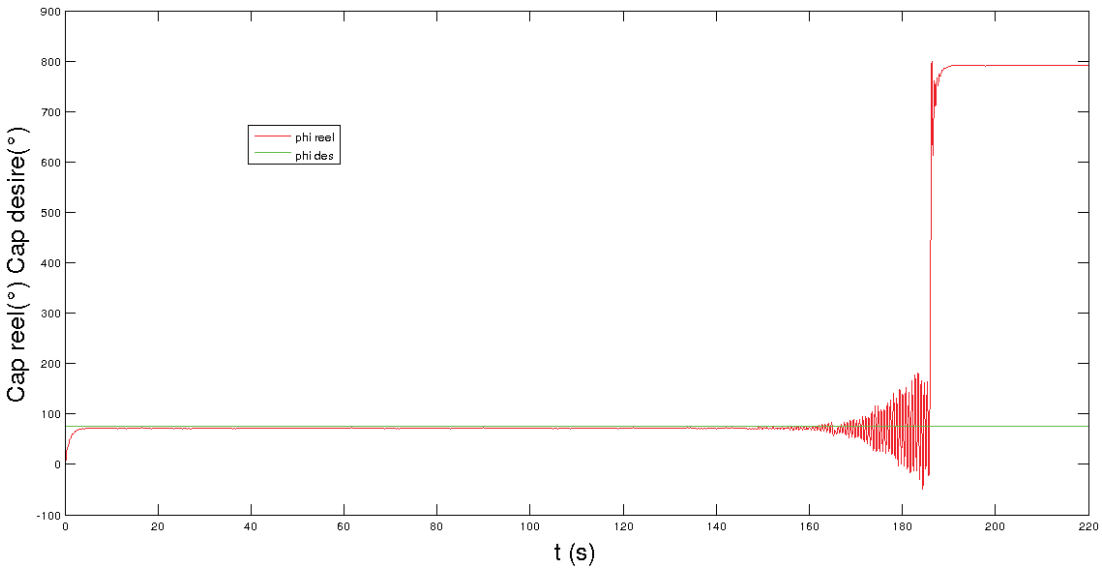

FIGURE 9.8 – A cause du non respect des *Contraintes* temporelles, l'asservissement en cap perd sa stabilité

les contraintes fixées avec des deltas respectifs de 482ms, 218ms et 518.2ms. Si la première violation ne provoque pas d'instabilité, les deux suivantes entraînent le comportement observé Figure 9.8. La raison de ce comportement est à trouver dans la manière dont nous réalisions les logs de données jusque là. En effet, les données d'exécution des modules (entrées, sorties, durées d'exécution) étaient stockées dans des fichiers dans la mémoire de la BeagleBone. En outre, le module ordonnanceur de ContrACT loggait lui aussi ses données d'exécution (date d'activation des différents modules, date d'arrêt de leur exécution, apparition de retards, préemptions).

Lorsque les données doivent être écrites dans un fichier (en utilisant la fonction *fprintf*), elles sont d'abord bufferisées avant d'être effectivement écrites dans le fichier une fois le buffer plein. Or, c'est cette écriture mémoire qui, sur la BeagleBone, pose problème. En effet, sa mémoire utilisée pour stocker les données est une mémoire eMMC. L'écriture dans une mémoire de ce type est non préemptible, c'est-à-dire qu'une fois démarrée, nos modules doivent attendre que l'écriture soit terminée avant de pouvoir reprendre leur exécution.

Nous avons donc décidé de changer notre manière d'effectuer les logs de données afin d'éviter ce problème. Pour tous les modules applicatifs, nous avons choisi d'utiliser un module dédié, chargé de collecter les données d'exécution des autres modules puis de les transmettre au PC de supervision où elles seront stockées dans un fichier. Nous avons également désactivé le log réalisé par le module ordonnanceur afin qu'il ne perturbe pas l'exécution nominale de notre application tout en nous autorisant sa réactivation si besoin est.

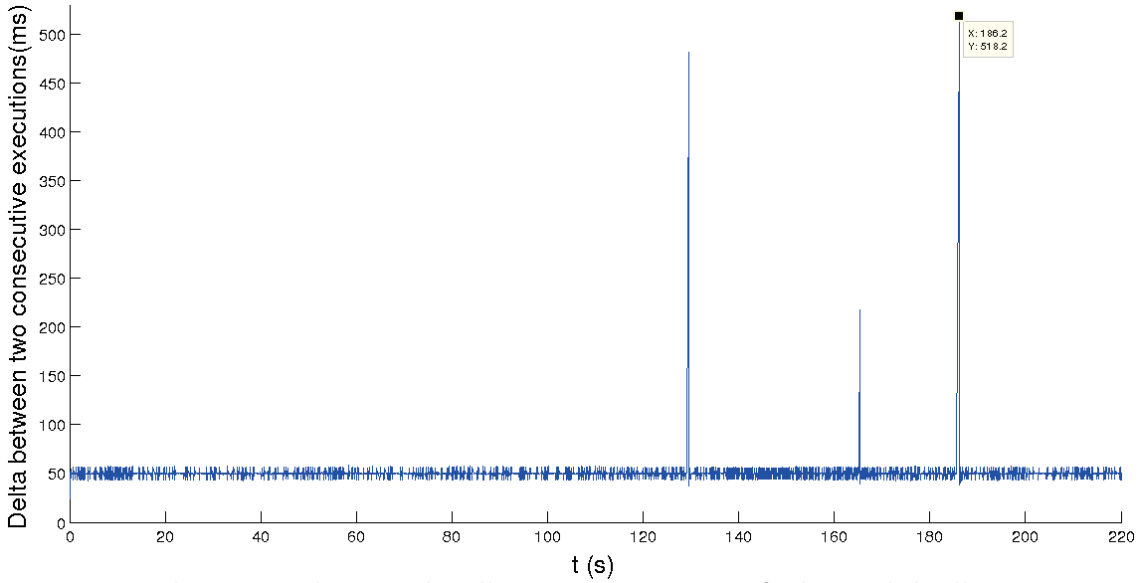

FIGURE 9.9 – Délais entre deux cycles d'exécution consécutifs du module d'asservissement en cap

9.2.2 Simulation avec contraintes respectées

Une fois ces modifications réalisées, l'exécution de notre application va respecter les *Contraintes* temporelles fixées. Un exemple de simulation est proposé Figure 9.10.

Nous avions pour coefficients $Kp = 10$, $Kd = 10$ et $KI = 0$ avec un bruit de 0.1 rad sur la mesure de cap et de 0.05 $rad.s^{-1}$ sur la mesure de vitesse angulaire.

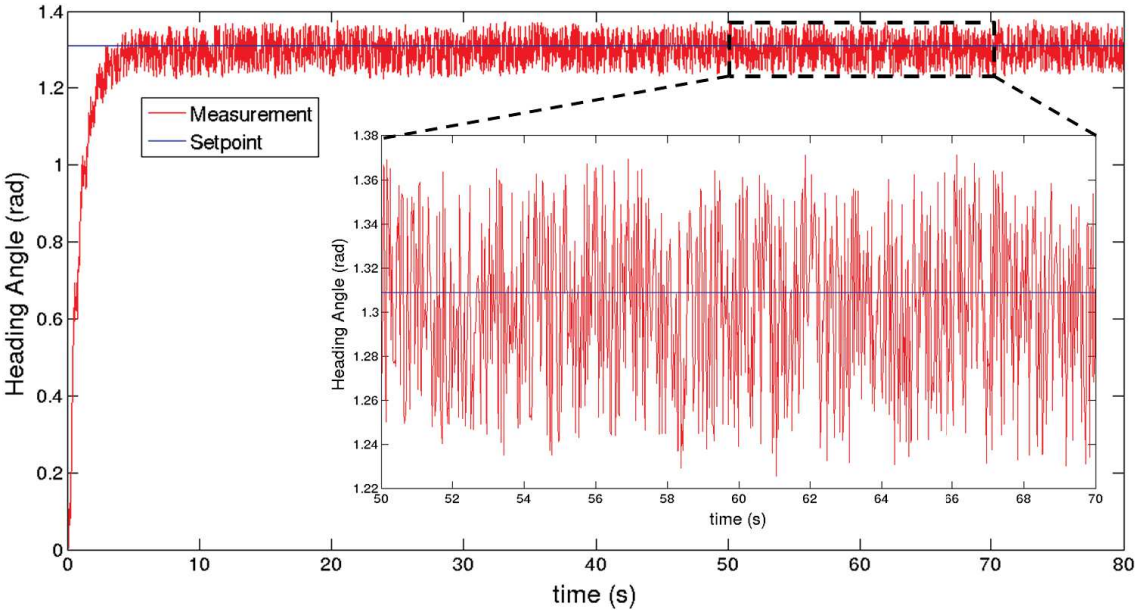

FIGURE 9.10 – Asservissement en cap avec *Contraintes* temporelles respectées.

Pour implémenter notre *Composition*, nous avons utilisé 6 modules ContrACT. Le premier *Sensors* regroupe la réception des données capteurs (IMU, Loch Doppler et capteur de profon-

deur) provenant du simulateur et le second *Actuators* gère l'envoi des consignes actionneurs au simulateur. Le module *MDynamic* contient le modèle dynamique et enfin *Navigation* contient la navigation du robot (ici simplifiée puisqu'elle ne contient que la partie relative à l'asservissement en cap). Ces quatre modules sont regroupés dans un schéma, nommé $C1$, fonctionnant à une période de 25ms. L'asservissement en cap est mis en œuvre dans le module *Control* et le changement de repère est effectué dans le module $FrameShift$, les deux étant regroupés dans un schéma, nommé $C2$, fonctionnant à une période de 50ms. Nous présentons dans les deux figures suivantes, les écarts entre le début effectif de chaque cycle et le début théorique de celui-ci basé sur la période d'exécution de chaque module. Ainsi, le délai, pour un cycle i, est calculé comme :

$$delai_i = t_{start\_the_i} - t_{start\_real_i} \tag{9.29}$$

avec :

$$t_{start\_the_i} = t_{start\_real_0} + i * period \tag{9.30}$$

Nous voyons clairement l'apparition de délais dûs aux temps de calculs variables au sein des modules et à l'exécution de schémas à des périodes différentes (nous verrons plus loin que, lorsque tous les modules s'exécutent à la même période, l'évolution de ces retards est beaucoup plus régulière). Néanmoins, l'ordonnanceur arrive à le compenser afin de les garder dans une proportion acceptable vis à vis des périodes d'exécution, comme le montre le Tableau 9.1. Le module le plus affecté par les erreurs est le modèle dynamique. En effet, il s'agit d'un des plus lourds du point de vue calculatoire, ce qui le rend plus délicat à placer pour l'ordonnanceur.

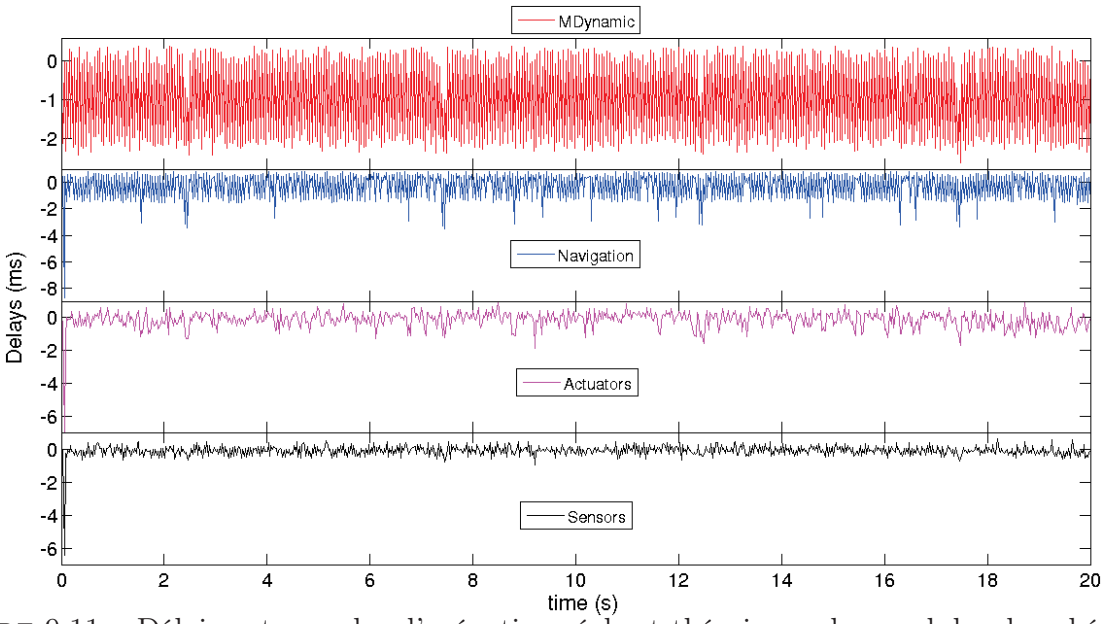

FIGURE 9.11 – Délais entre cycles d'exécution réels et théoriques des modules du schéma $C1$

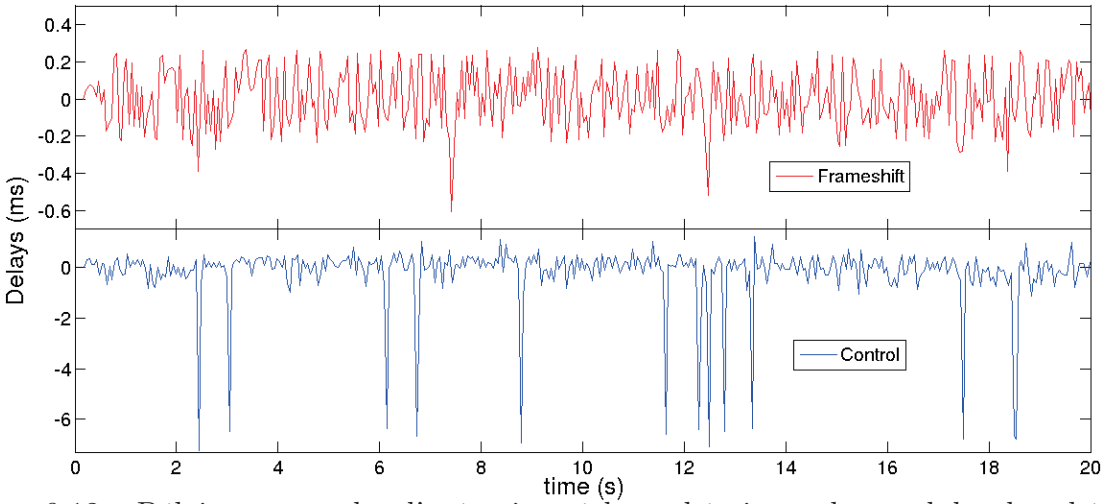

FIGURE 9.12 – Délais entre cycles d'exécution réels et théoriques des modules du schéma $C2$

Tableau 9.1 – Délais moyens entre les périodes d'exécution théoriques et effectives des modules

Module	Schéma	Délai moyen(ms)	Période(ms)	Erreur(%)
Sensors	C1	-0.3583	25	1.433
Actuators	C1	-0.1535	25	0.614
Navigation	C1	-0.2808	25	1.123
MDynamic	C1	-1.0358	25	4.143
Control	C2	-0.4878	50	0.976
Frameshift	C2	-0.0173	50	0.035

La Figure 9.13 et la Figure 9.14 présentent les durées d'exécution cumulées des modules contenus dans les schémas $C1$ et $C2$ respectivement. Comme nous pouvons le constater, la variabilité de ces durées d'exécution est importante. En effet, de nombreux phénomènes tels que les préemptions peuvent venir influencer ces durées. Cela souligne la difficulté d'estimation du pire temps d'exécution (*propriété* $t_{comp_{max}}$) d'un *Atome* et l'apport de la simulation HIL, pour ajuster ces données. Néanmoins, dans notre cas, nous voyons que ces durées restent très inférieures à la période d'exécution des différents schémas garantissant le respect de nos *Contraintes* temporelles.

9.3 Le problème de l'étage d'actionnement

Nous allons maintenant nous intéresser à un problème expérimental lié à la grande disparité des caractéristiques des actionneurs de notre robot. Nous allons voir comment son impact n'a

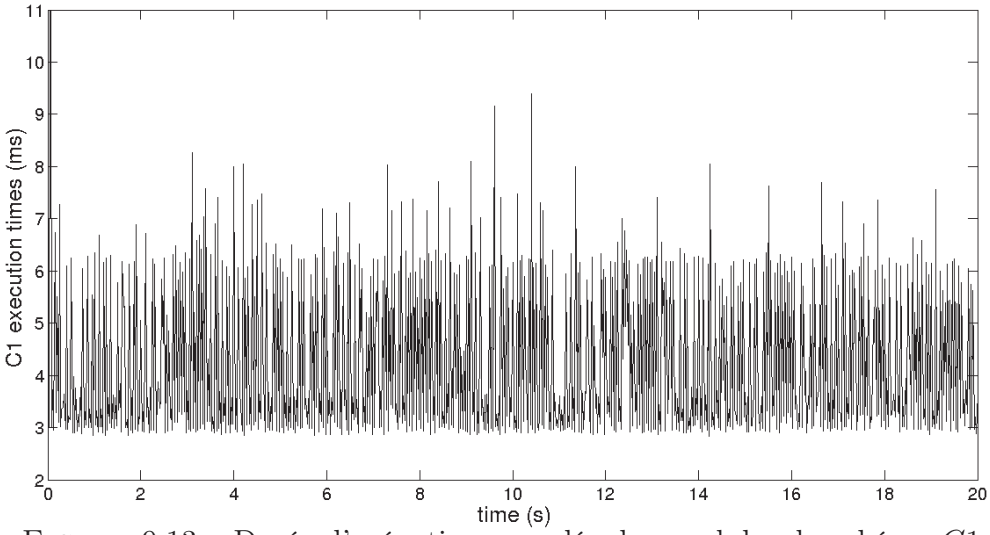

FIGURE 9.13 – Durée d'exécution cumulée des modules du schéma $C1$

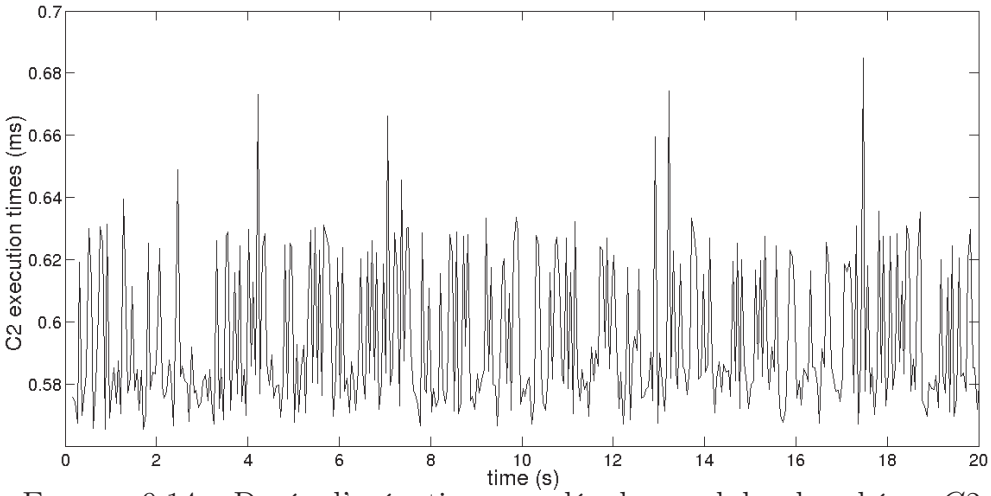

FIGURE 9.14 – Durée d'exécution cumulée des modules du schéma $C2$

pu être apparent que lors des expérimentations sur le robot réel et comment, grâce à la structuration permise par les *Atomes*, nous avons pu modifier efficacement notre *Composition* afin de corriger l'impact de ces caractéristiques.

Dans cet exemple, les limitations de fonctionnement de l'IMU et des différents capteurs font que l'ensemble de nos *Atomes* mis en œuvre dans les modules ContrACT fonctionneront à une période de 100ms.

9.3.1 Une première expérimentation, le problème des actionneurs

La Figure 9.15 expose les résultats d'expérimentation de l'asservissement en cap sur la version 6 moteurs (version de base) du Jack lors de la réalisation d'un mouvement d'avance. Comme nous pouvons le constater, le comportement du robot n'est absolument pas celui attendu. Nous avions pour paramètres du PID $Kp = 1$, $Kd = 1$ et $KI = 0$. Nous voyons

l'apparition de deux phénomènes : un important offset ainsi que des oscillations.

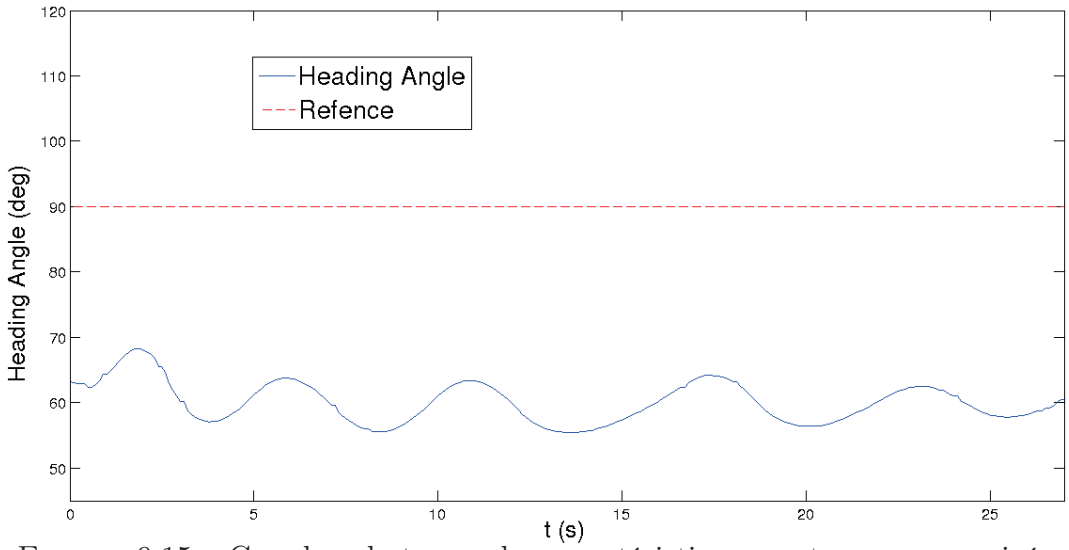

FIGURE 9.15 – Cap du robot avec des caractéristiques moteurs non corrigées

Ce problème vient des actionneurs. En effet, nous avions, lors de la conception de la *Molécule Actuation* (Section 9.1.3), supposé que ces moteurs avaient une caractéristique globalement linéaire et que les différents moteurs étaient identiques.

Or, la Figure 9.16 montre les caractéristiques (i.e. la relation $Fp = f(PWM)$) des quatre moteurs utilisés pour les déplacements dans le plan horizontal et donc utilisés pour l'asservissement en cap. Il faut en premier lieu noter :

- La zone linéaire occupe une portion très faible de la caractéristique
- La présence d'une importante zone morte qui était en partie responsable des oscillations
- La disparité entre les différentes caractéristiques
- Les moteurs ne sont pas réversibles

En outre, les moteurs arrière gauche et avant droit ont le même pas, qui est différent de celui des moteurs arrière droit et avant gauche (nécessaire pour que les couples générés par la rotation des moteurs se compensent), ce qui accentue les différences de caractéristiques entre les moteurs. Ainsi, en certains endroits de la caractéristique, les différences de poussée, pour un même PWM de commande, atteignent près de 15%.

Ainsi, si l'on applique la même consigne aux quatre moteurs, dans ce cas $PWM = 50$, le robot devrait rester immobile. Or, comme le montre la Figure 9.17, la disparité des caractéristiques moteurs provoquent une importante rotation.

De fait, la disparité est telle que même l'utilisation d'un asservissement n'est pas suffisant pour contrer complètement le phénomène.

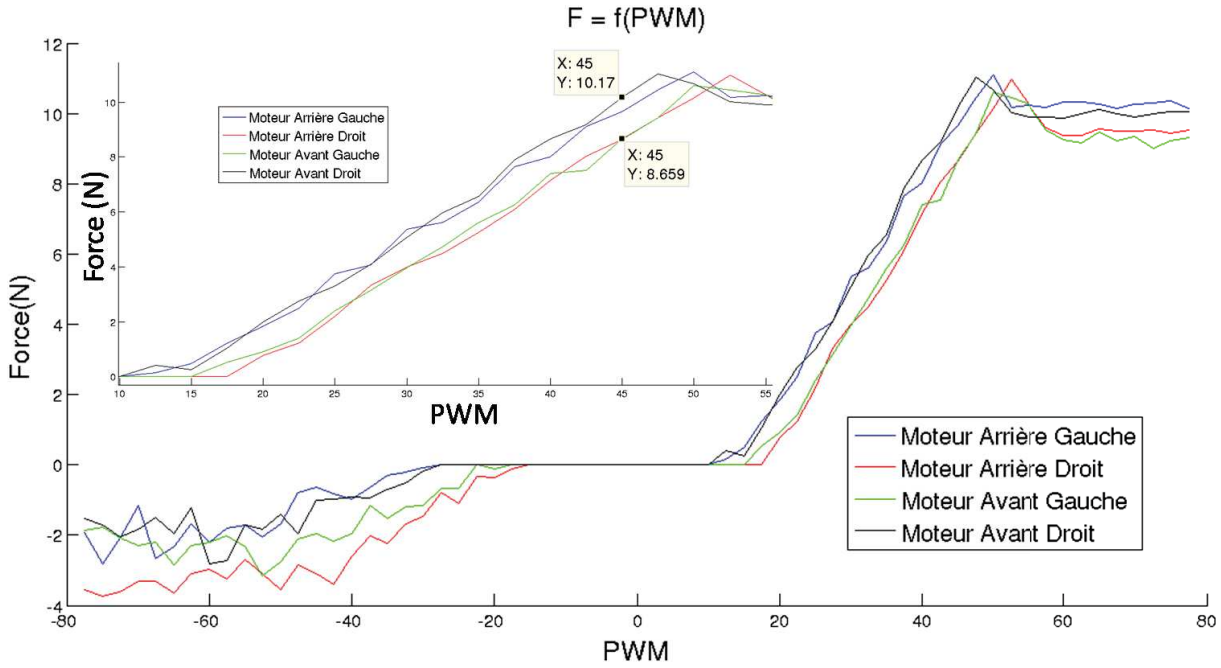

FIGURE 9.16 – Caractéristiques des quatre moteurs situés dans le plan horizontal du Jack version six moteurs

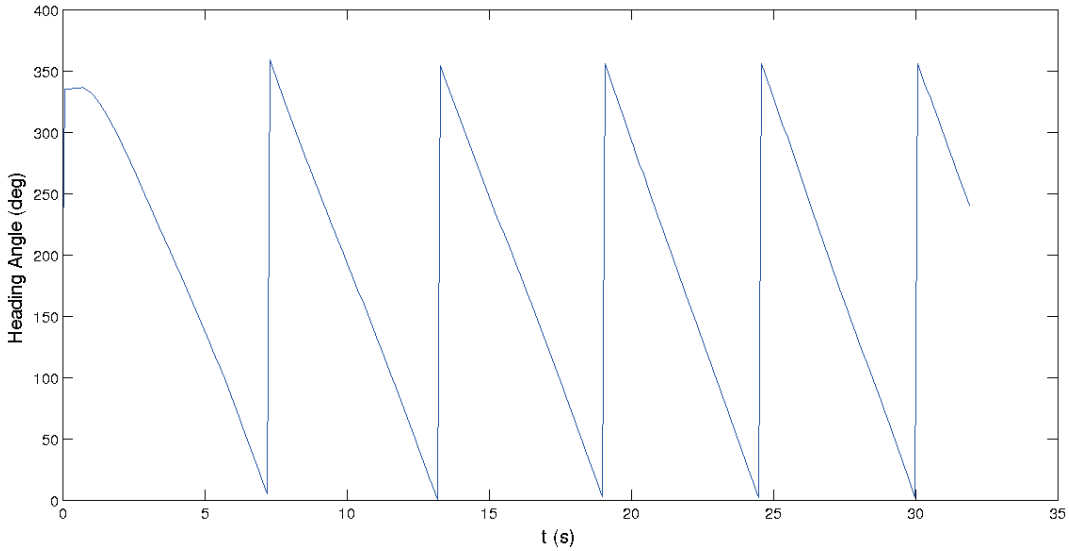

FIGURE 9.17 – Cap du robot avec un régime moteur identique pour les moteurs du plan horizontal

9.3.2 Modification de la Molécule Actuation

Pour pallier à ce problème, nous allons modifier la *Molécule Actuation* que nous avons présentée Section 9.1.3. Grâce à notre décomposition, nous savons que la partie à modifier se situe au niveau de la *Molécule InverseEngineCaracteristics* (Figure 9.7). Notre objectif va donc être de modifier notre *Composition* afin de **replacer les forces demandées aux quatre moteurs** dans la **partie linéaire de leurs caractéristiques**, ce que nous permet

la redondance d'actionnement [RLLA15]. La Figure 9.18 présente la version modifiée de cette partie de la *Molécule*, les entités modifiées ou ajoutées pour la correction apparaissent en rouge.

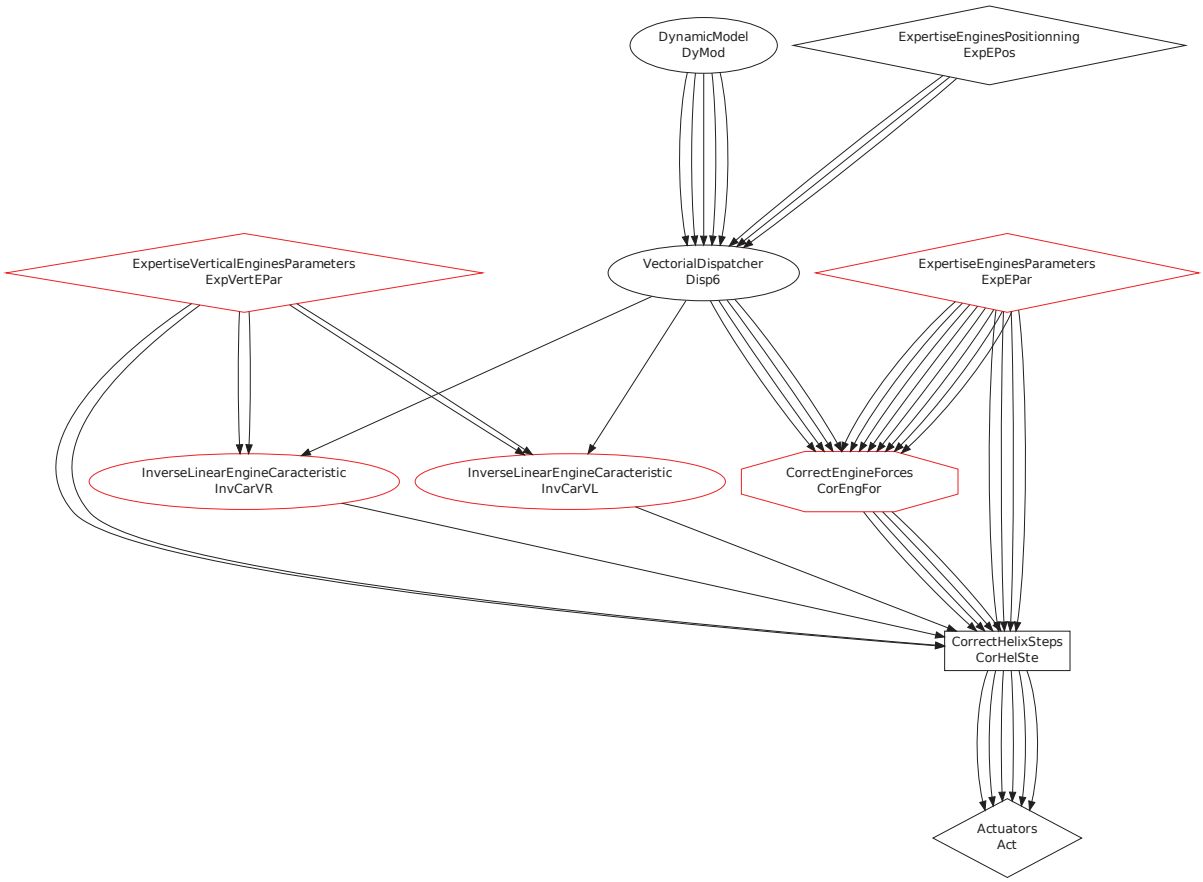

FIGURE 9.18 – *Graphe Moléculaire* de la *Molécule Actuation* (deuxième partie) modifiée pour prendre en compte le problème des caractéristiques moteurs

Pour les moteurs verticaux, l'utilisation de la caractéristique n'a pas été modifiée car de par le positionnement des moteurs et la forme du robot, l'impact d'une mauvaise évaluation de la caractéristique de ces moteurs est limité lors de ces déplacements verticaux (cela ne cause par exemple pas d'instabilité comme observé pour l'asservissement en cap). Nous continuons donc à utiliser deux *Instances* de l'$AtomeInverseLinearEngineCaracteristic$, $InvCarVL$ pour le moteur vertical gauche et $InvCarVR$ pour le vertical droit. Les caractéristiques des moteurs et le pas d'hélice associé sont valués par la *Connaissance Externe* $ExpertiseVerticalEnginesParameters$ que nous avons séparée de celle valuant les propriétés des moteurs situés dans le plan horizontal pour accroître la modularité.

L'*Alternative* $CorrectEngineForces$ assure la gestion des moteurs situés dans le plan horizontal. Les différents paramètres des moteurs (coefficients a et b de la caractéristique de chaque moteur et pas de chaque hélice) sont fournis par la *Connaissance Externe*

ExpertiseEnginesParameters.

CorrectEngineForces doit effectuer différentes opérations afin de corriger les problèmes émanant à la fois de la disparité des caractéristiques moteurs et de la faible plage de PWM située dans la zone linéaire de cette caractéristique. Le contenu de l'*Alternative* est donné Figure 9.19. Il est à noter que dans cette figure et les suivantes, nous ne représenterons que les paramètres d'un seul moteur (le moteur avant gauche) afin d'alléger les figures pour en simplifier la lecture.

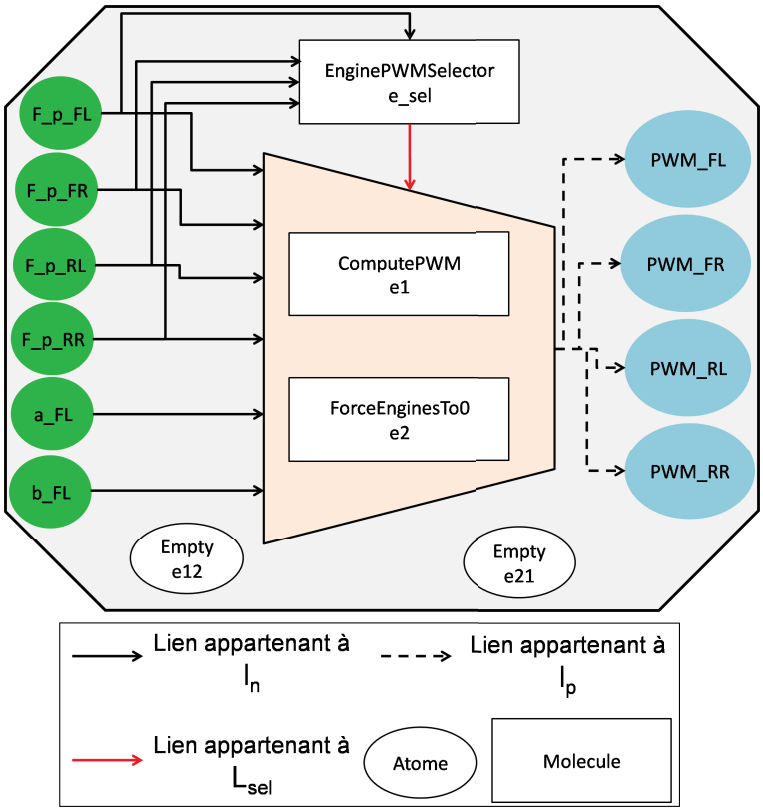

FIGURE 9.19 – *Alternative CorrectEngineForces*

Il faut tout d'abord savoir si des consignes sont transmises aux moteurs (i.e. si au moins une *Fp* est non nulle). Si c'est le cas, nous allons forcer les sorties des quatre PWM à zéro via la *Molécule ForceEnginesTo*0. En effet, si nous n'effectuons pas cela, notre correction va faire tourner les moteurs à une vitesse minimale entraînant une consommation d'énergie inutile. En outre, comme les moteurs se retrouveront en limite de zone non linéaire, ils ne seront pas parfaitement équilibrés et le robot aura tendance à effectuer de légers déplacements.

C'est l'*Entité Composable* de sélection, *e_sel* qui est chargée de choisir si les consignes moteur doivent être mises à zéro ou si nous devons utiliser la méthode de calcul décrite plus loin. Il s'agit d'une *Molécule* décrite Figure 9.20. Elle reprend la structure habituelle d'un processus décisionnel tel que, par exemple, décrit dans le cadre du contrôle avec commutations (Section 2.2.1). Elle est ainsi constituée de deux *Atomes*. Le premier, *EnginesActive*, est

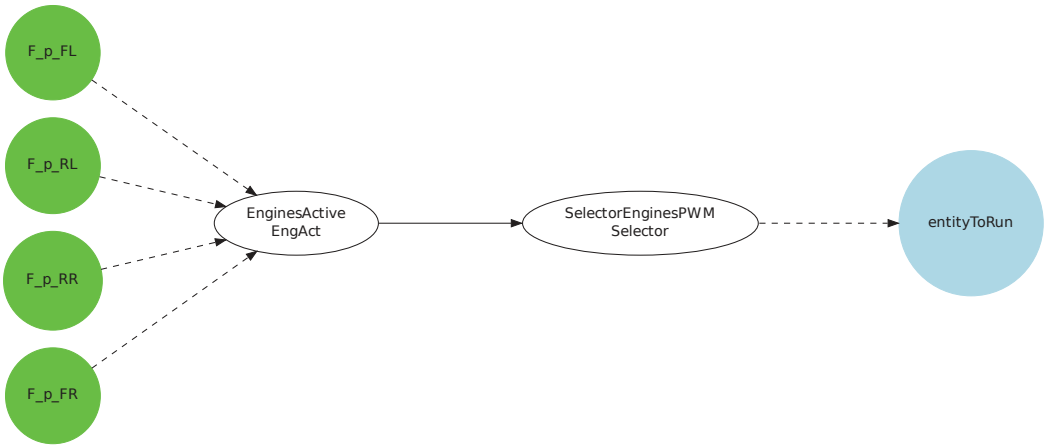

FIGURE 9.20 – *Molécule EnginePWMSelector*

chargé d'observer l'état du système. Dans ce cas, il doit déterminer si au moins un moteur est actif (i.e. la force qu'il doit produire est non nulle). Sa *Physique* est ainsi définie comme :

$$
\begin{aligned}
&if(F\_p\_FL \neq 0 \ or \ F\_p\_FR \neq 0 \ or \ F\_p\_RL \neq 0 \ F\_p\_RR \neq 0)\\
&\{\\
&\quad active = true;\\
&\}\\
&else\\
&\{\\
&\quad active = false;\\
&\}
\end{aligned}
$$

Le second est un *Atome* de la famille des Sélecteurs (voir Annexe B, Exemple B.8 pour plus de détails) qui est chargé de représenter le processus décisionnel basé sur ces observations et de choisir l'entité à exécuter. Sa *Physique* est définie comme :

$$
\begin{aligned}
&if(active == true)\\
&\{\\
&\quad entityToRun = "e1";\\
&\}\\
&else\\
&\{\\
&\quad entityToRun = "e2";\\
&\}
\end{aligned}
$$

Les deux *Entités Composables* substituables étant simples, nous utiliserons les *Atomes Empty* pour indiquer qu'aucune opération n'est nécessaire lors de la jonction. Pour une explication plus détaillée de l'utilité de cet *Atome*, le lecteur est invité à se référer à l'Annexe B, Exemple B.8.

Enfin, si les moteurs sont actifs, nous allons utiliser la *Molécule ComputePWM* pour déterminer les valeurs de PWM à appliquer aux moteurs. Comme nous l'avons vu, la plage de la zone linéaire est limitée. Il nous faut donc nous assurer que les commandes à envoyer aux moteurs se situent dans cette plage. La *Molécule* est décrite Figure 9.21.

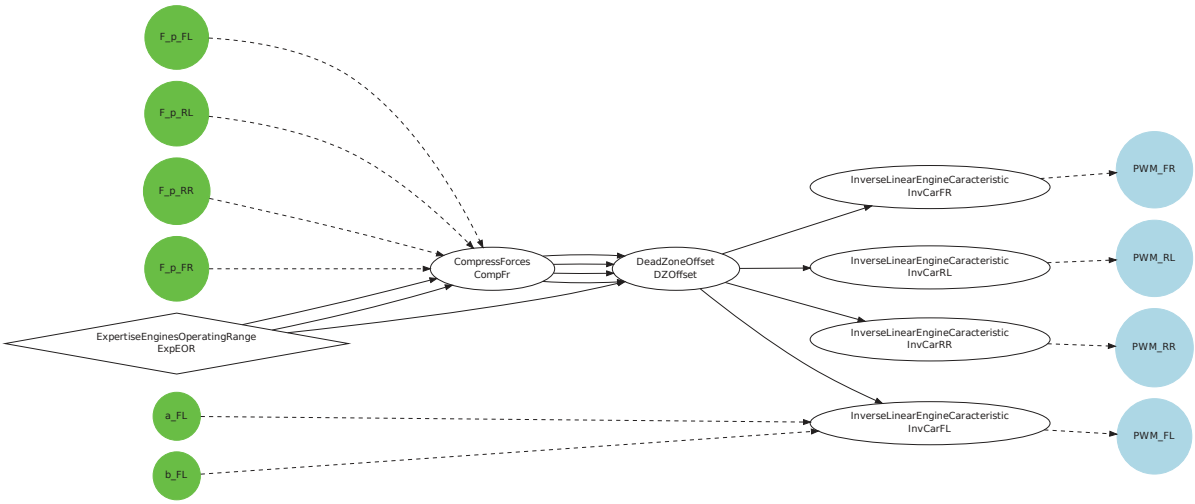

FIGURE 9.21 – *Molécule ComputePWM*

C'est l'*Atome ExpertiseEnginesOperatingRange* qui fixe, ici, la plage de fonctionnement linéaire des moteurs sous la forme d'une force minimale et maximale ($F_{OR_{min}}$ et $F_{OR_{max}}$). Il faut ainsi noter que le domaine d'utilisation de notre *Molécule* se limite au cas où la zone linéaire est cantonnée aux PWM et forces positifs. La première étape est de compresser les forces actionneurs pour qu'elles tiennent dans la plage linéaire.

Ainsi si nous avons :

$$F_{act_{min}} = min(F\_p\_FL, F\_p\_RL, F\_p\_FR, F\_p\_RR) \tag{9.31}$$

$$F_{act_{max}} = max(F\_p\_FL, F\_p\_RL, F\_p\_FR, F\_p\_RR) \tag{9.32}$$

Nous allons compresser les forces, dans l'*Atome CompressForces*, de manière à ce que :

$$F_{act_{max}} - F_{act_{min}} \leq F_{OR_{max}} - F_{OR_{min}} \tag{9.33}$$

Pour cela, si la condition (9.33) n'est pas vérifiée, nous allons diviser les quatre forces par le coefficient :

$$\frac{F_{OR_{max}} - F_{OR_{min}}}{F_{act_{max}} - F_{act_{min}}} \tag{9.34}$$

Nous allons ensuite, grâce à l'*Atome* $DeadZoneOffset$, nous assurer que le moteur poussant le moins soit dans la zone linéaire de la caractéristique, c'est-à-dire que :

$$F_{act_{min}} \geq F_{OR_{min}} \tag{9.35}$$

Sinon, nous allons ajouter à chaque force le terme :

$$F_{OR_{min}} - F_{act_{min}} \tag{9.36}$$

Ainsi toutes les forces se retrouvent dans la zone linéaire de la caractéristique et nous appliquons enfin les caractéristiques inverses identifiées pour chaque moteur.

Il nous faut également noter que si cette méthode améliore grandement le pilotage du robot, elle présente certaines limitations. Premièrement, tous les moteurs poussent dans le sens positif. Or par exemple, pour un mouvement d'avance ($u > 0$), les moteurs arrières doivent exercer une force positive mais la force positive exercée par les moteurs avant va s'opposer au mouvement. Cela entraîne une surconsommation d'énergie. Néanmoins pour résoudre ces problèmes, nous n'avons d'autre solution que de remplacer les moteurs du robot par d'autres plus performants et qui seraient, entre autres, réversibles, c'est-à-dire utilisables avec une poussée négative.

Comme illustré à travers cet exemple simple, nous voyons bien l'apport de la décomposition en *Atomes* lorsque des modifications doivent être effectuées dans la description d'une fonctionnalité robotique. En effet, l'expressivité obtenue par l'explicitation de l'apport de chaque entité permet de faciliter la tâche du concepteur dès qu'il doit faire évoluer une *Composition* puisque les entités impactées sont clairement identifiées.

9.3.3 Résultats expérimentaux

La Figure 9.22 présente les résultats de l'asservissement en cap une fois prises en compte les caractéristiques des moteurs présentées précédemment. Trois courbes sont proposées. La courbe verte (étoiles) représente les résultats de l'asservissement en cap sans l'utilisation de la correction des caractéristiques des moteurs. La courbe bleue (ronds) utilise la correction présentée précédemment mais conserve des caractéristiques identiques pour les moteurs. Nous pouvons voir que l'offset a été réduit tout comme l'amplitude des oscillations. La courbe noire représente la correction utilisant les paramètres identifiés pour chaque caractéristique. Nous

voyons que l'offset a été fortement réduit tout comme les oscillations. Toutefois, notre estimation des caractéristiques moteurs reste perfectible mais identifier la caractéristique exacte demeure difficile d'autant que d'autres phénomènes (tel qu'une orientation imparfaite des propulseurs sur le robot) peuvent légèrement altérer la réponse des moteurs.

Pour pallier aux limitations de cette correction, nous avons mis en place un autre correcteur. Pour plus de détails sur celui-ci, le lecteur est invité à se référer aux travaux de Benoit Ropars (Benoit Ropars, *Un vecteur robotique polyvalent pour l'exploration sous-marine faible fond*, thèse soutenue le 16 décembre 2015 à Montpellier, LIRMM).

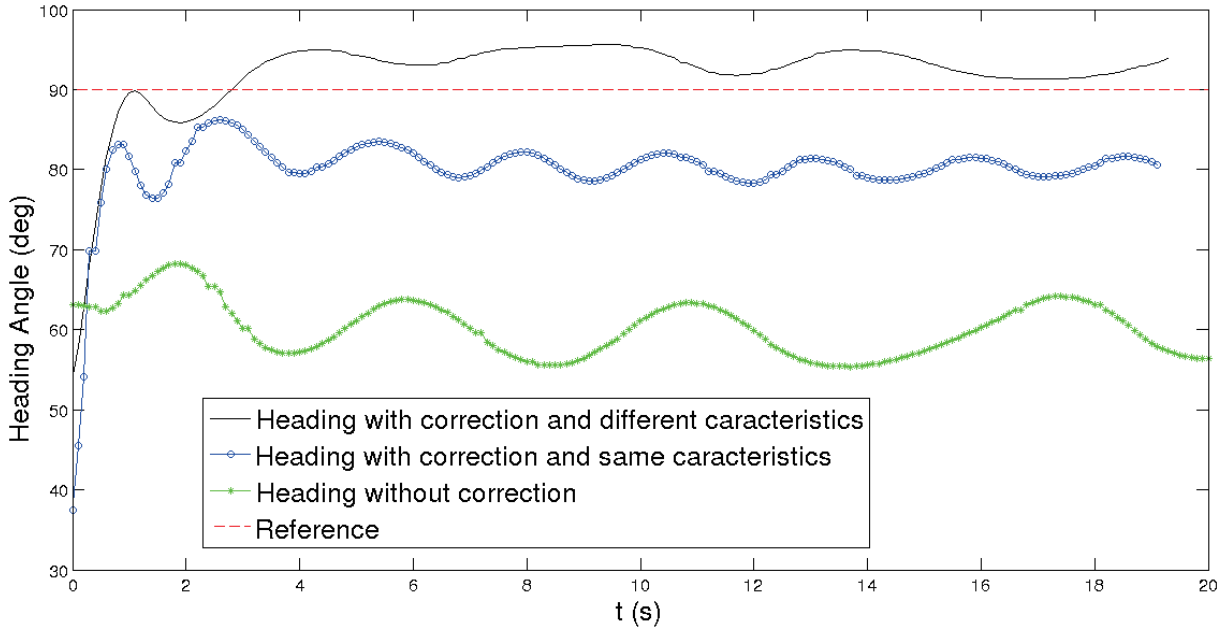

FIGURE 9.22 – Asservissement en cap avec différents niveaux de correction

La Figure 9.23 présente plusieurs cycles d'exécution des différents modules utilisés dans notre architecture. Nous pouvons voir que les différents cycles s'exécutent bien à la période de 100ms désirée. De plus, comme nous l'avons déjà évoqué Section 9.2.2, le fait que tous les modules s'exécutent à la même période apporte beaucoup plus de régularité à l'ordonnancement et l'ordre des modules n'est pas modifié d'un cycle à l'autre. Les ronds représentent les dates de début et les croix les dates d'arrêt de chaque module. Les délais entre le démarrage et l'arrêt d'un module sont à la fois imputables à l'exécution de l'ordonnanceur et aux *threads* qui sont utilisés pour gérer la communication avec certains capteurs (principalement lorsque celle-ci est bloquante) et qui ne sont pas représentés ici. Nous pouvons également voir que nos calculs durent moins que la période souhaitée assurant du respect des *Contraintes* temporelles.

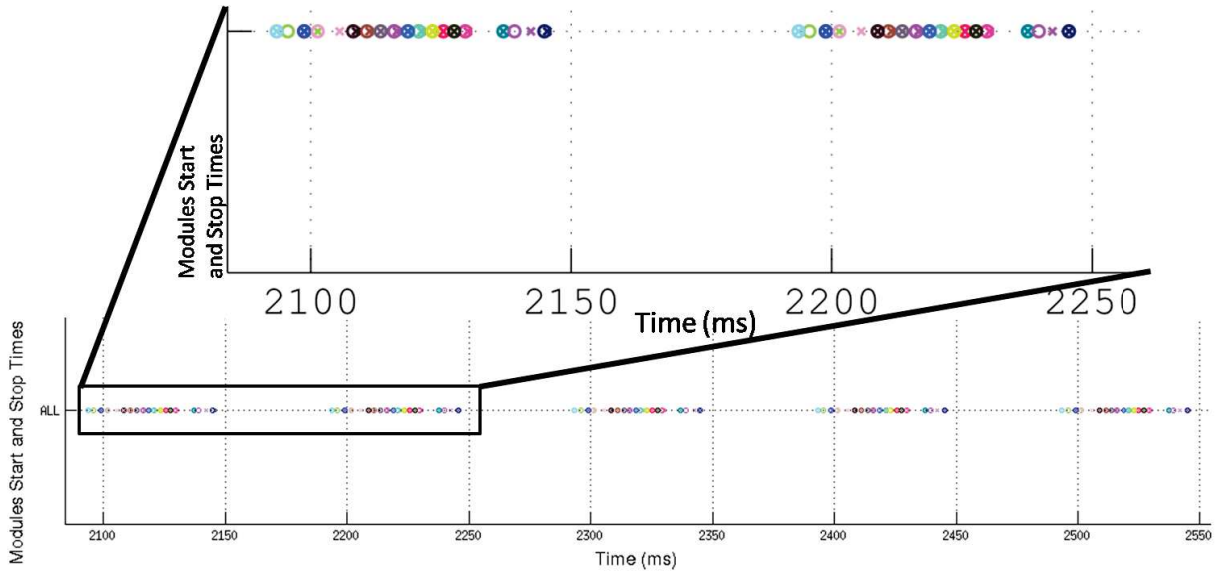

FIGURE 9.23 – Date de début (ronds) et de fin (croix) d'exécution des différents modules utilisés pour implémenter notre fonctionnalité

9.4 Exemples Applicatifs

Nous allons ici présenter différentes applications réalisées avec le Jack en version 6 moteurs (souvent équipé de son *skid*). Lors de toutes ces applications, nous avons utilisé la *Composition* décrite au début de ce chapitre.

9.4.1 Inspection d'une paroi

Une première application effectuée était l'inspection d'une paroi réalisée dans le port de Palavas-les-Flots afin de tester notre caméra acoustique. Pour cela nous avons positionné le robot face à la paroi et parcourons celle-ci avec un déplacement latéral. La Figure 9.24 montre le robot en train de suivre la paroi et la Figure 9.25 montre le retour du sonar sectoriel. Nous distinguons les différents rochers dont est constituée la paroi. La précision du retour du sonar sectoriel nous a incités à démarrer des travaux pour l'intégrer logiciellement à notre application afin d'asservir le suivi de paroi pour simplifier le pilotage du robot.

La Figure 9.26 montre le cap suivi par le robot lors du suivi de paroi. Nous observons un léger offset et des oscillations plus importantes que celles observées en régime statique (voir Figure 9.22, courbe noire). Cela s'explique par les perturbations induites par l'ombilical et également par le *skid* comme expliqué au Chapitre 8.

9.4.2 Inspection d'une épave

Nous avons également emmené le robot dans un cimetière sous-marin près de la ville de Sète. Il s'agit notamment d'un site d'intérêt pour les biologistes car les différentes épaves servent d'abris artificiels à de nombreuses espèces marines comme le montre la Figure 9.27.

FIGURE 9.24 – Inspection d'une paroi constituée de rochers dans le port de Palavas-les-Flots

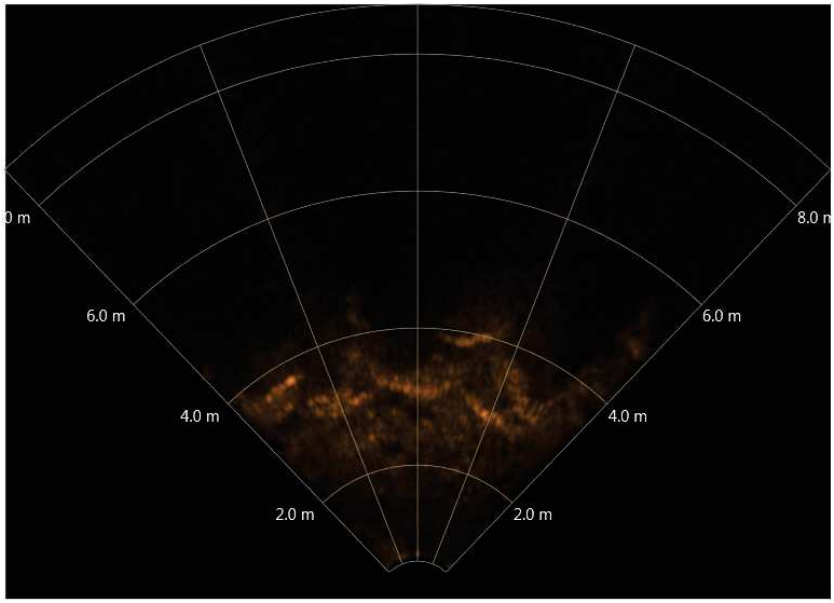

FIGURE 9.25 – Rendu du sonar sectoriel sur lequel nous distinguons les différents rochers

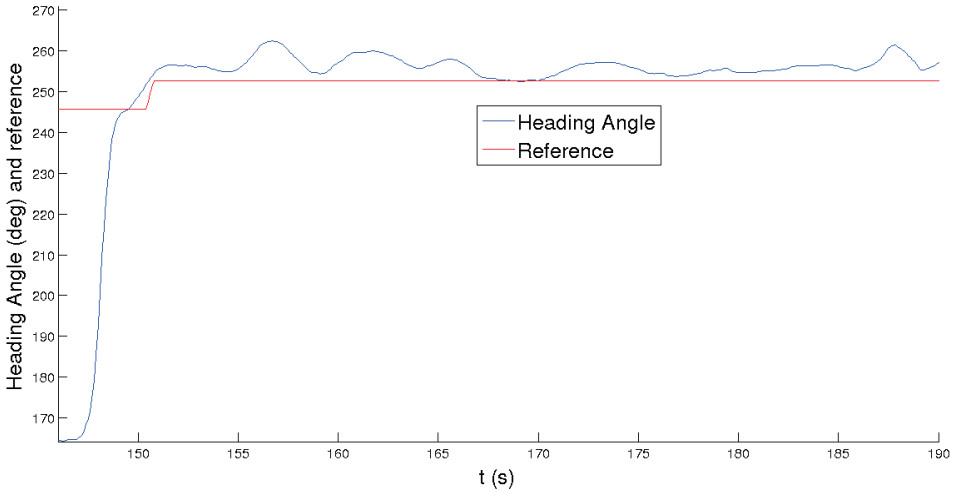

FIGURE 9.26 – Cap du robot lors de l'inspection de la paroi

FIGURE 9.27 – Observation d'un hippocampe (encadré en rouge)

Nous avons également réalisé un scan d'une épave située près du ponton comme montré Figure 9.28. La Figure 9.29 montre un des scans effectués. Nous distinguons la cabine et la coque du bateau au-dessus du fond. Néanmoins, comme notre robot avait tendance à être poussé sur la gauche par les courants et l'effet de traction du câble, la partie droite de l'épave était occultée. Ainsi, des asservissements supplémentaires en vitesse sont nécessaires afin de contrer les perturbations environnementales mais nécessitent l'utilisation du Loch Doppler.

9.4.3 Cartographie d'une berge d'un canal

Nous terminons ce chapitre par la présentation d'une application de cartographie d'une berge du Canal du Midi pour évaluer l'état des berges (qui subissent une érosion induite par la motorisation des bateaux circulant sur le canal). Si elle est réalisée dans un environnement moins complexe et donc donnant plus de marge d'erreur qu'un aquifère karstique, cette application présente certaines similitudes avec la cartographie d'aquifères karstiques. En effet, elle

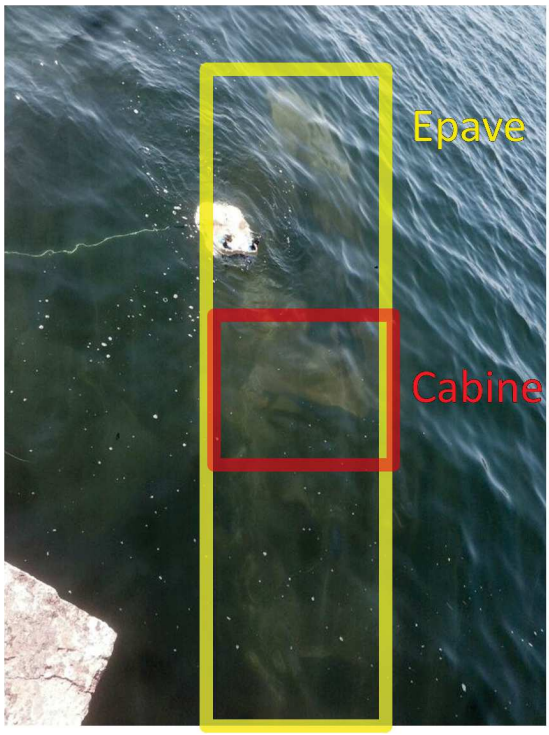

FIGURE 9.28 – Le robot passe au-dessus d'une épave. Celle-ci, difficilement visible depuis la surface, est entourée en jaune

nécessite d'utiliser un capteur proximétrique, dans notre cas un sonar profilométrique, afin de réaliser la cartographie et aussi d'assurer la sécurité du robot (dans ce cas, il s'agit d'éviter les berges du canal).

Nous avons, dans le cadre de cette expérimentation, utilisé la stratégie de commande présentée au début de ce chapitre afin d'effectuer la cartographie. Nous n'avons par conséquent pas mis en œuvre de stratégie visant à éviter les berges du canal.

Néanmoins, notre principal problème pour réaliser une reconstruction correcte a été d'éviter que le robot ne percute les parois. En effet, un écart même très minime entre la consigne de cap fournie par l'utilisateur et l'axe du canal va avoir tendance à faire se rapprocher le robot d'une des berges. Sans une fonctionnalité permettant de maintenir une distance constante à une berge ou de se centrer dans le canal (suivant les besoins de l'application), nous pouvons donc très difficilement effectuer la cartographie en toute sécurité et avons été obligé, comme le montre la Figure 9.30, de tenir le robot avec une perche pour maintenir une distance relativement constante à la berge sur toute la longueur de la cartographie.

Une section de cette reconstruction est proposée Figure 9.31. Les différentes portions repérées par un code couleur désignent les zones où la berge est érodée.

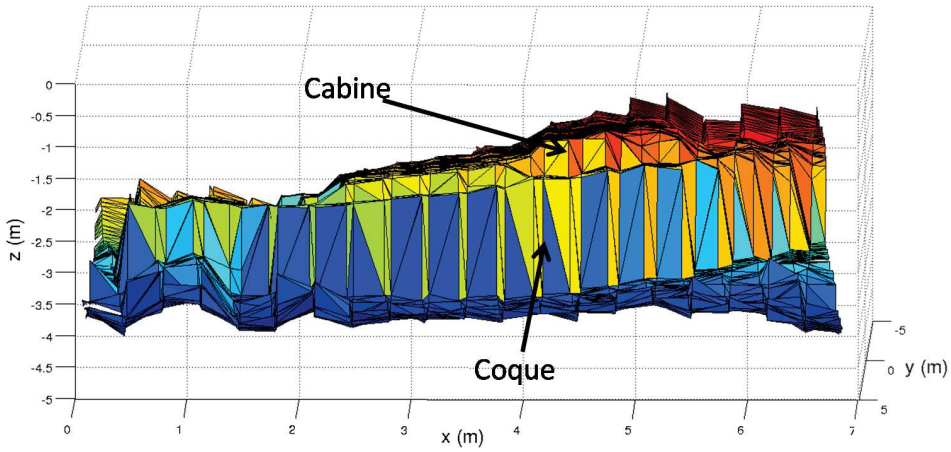

FIGURE 9.29 – Flan gauche de l'épave tel qu'obtenu suite à un scan sonar (résolution grossière)

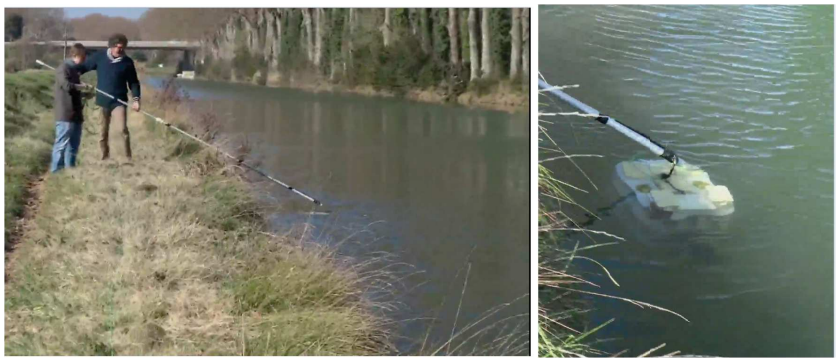

FIGURE 9.30 – Sans fonctionnalité de centrage, nous devons tenir le robot pour éviter qu'il ne percute une berge

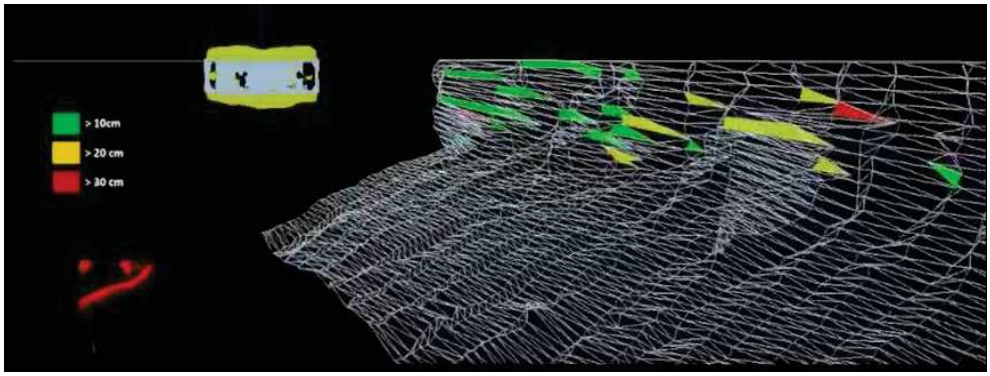

FIGURE 9.31 – Reconstruction d'une partie de la berge

9.5 Points clés du chapitre

- A travers une simulation *Hardware-in-the-Loop*, nous avons montré l'importance du respect des *Contraintes* temporelles fixées pour garantir la stabilité de notre contrôleur.
- Nous avons pu efficacement faire évoluer la *Molécule* chargée de gérer l'actionnement du robot grâce à la décomposition qui réifie le rôle de chaque entité utilisée et des interactions entre elles.
- Notre vecteur robotique a été mis en œuvre au travers de différentes expérimentations.

Chapitre 10

Application à l'évitement de parois dans un environnement karstique

Nous allons maintenant nous intéresser à une fonctionnalité du Jack version 12 moteurs, celle de l'évitement de parois pour l'exploration d'environnements karstiques. En effet, comme nous l'avons présenté au Chapitre 1, il s'agit d'environnements très complexes et confinés où la marge de manœuvre et d'erreur est très limitée. De fait, seulement contrôler le robot à l'aide d'asservissements comme celui présenté au chapitre précédent n'est pas possible. Nous avons d'ailleurs vu à la fin du chapitre précédent que même dans un environnement moins complexe les seuls asservissements sont insuffisants. Il nous faut notamment mettre en place des fonctionnalités permettant d'assurer la sécurité du robot pendant que l'opérateur humain prendra les différentes décisions.

Nous allons donc nous baser sur l'exemple d'évitement de parois introduit au Chapitre 4. Nous illustrerons les différentes améliorations apportées à l'exemple (très simplifié) qui fut proposé dans ce cadre. Enfin, de par la complexité des environnements karstiques, il n'est évidemment pas possible d'expérimenter dans de tels milieux sans une solide validation expérimentale préalable dans des environnements offrant une marge de manœuvre plus importante.

10.1 Le problème de l'implémentation

Nous allons ici repartir de l'exemple d'évitement de parois qui fut décomposé sous forme de *Blocs* à la Figure 6.11. Nous allons commencer par valuer les *Contraintes* temporelles à utiliser dans les différentes formules de diffusion présentées dans les équations 6.5 à 6.17.

Ces valeurs sont présentées dans le Tableau 10.1 pour les durées d'exécution et périodes d'exécution désirées de la *Physique* des différents *Atomes* et dans le Tableau 10.2 pour les valeurs de temps d'échanges de données au sein des différents *Blocs*. Toutes les données sont

indiquées en millisecondes. Pour les *Atomes* dont les *Eléments d'Interface* sont de *Type Array*, tels que *RobToDVZ* par exemple, deux valeurs sont données. La première correspond à une taille de tableau de 100 (100 points sonar, asservissement en 3 dimensions) et la seconde à une taille de 2 (asservissement en 2 dimensions). Lorsque des *Atomes* ayant des *Besoins* de *Type Array* sont présents dans un bloc, nous indiquerons également les valeurs de t_{comm} pour des tailles de tableau de 100 et 2 (voir Tableau 10.2). Pour les *Atomes* de *Connaissance Externe*, les temps sont estimés par l'exécution des modules ContrACT dans lesquels les drivers sont utilisés (ils sont signalés par une étoile en exposant dans le tableau).

Tableau 10.1 – Valeurs des différentes *Contraintes* temporelles dans l'exemple d'évitement de parois

Atome	$t_{comp_{max}}$ [1] (ms)	T_{phy} (ms)
PS*	0.04 [2]	{2500, 4000, 4500, 5600, 7900, 10200} [3]
SonToRob.CylToCart	41.8811 / 4.2118	[0 ∞]
SonToRob.FS	126.714 / 5.3816	[0 ∞]
SonToRob.CartToCyl	48.5613 / 2.4634	[0 ∞]
RobToDVZ.CylToCart	41.8811 / 4.2118	[0 ∞]
RobToDVZ.FS	128.703 / 8.4995	[0 ∞]
RobToDVZ.CartToCyl	48.5613 / 2.4634	[0 ∞]
DVZ	101.721 / 2.4884	[0 ∞]
RFC	89.567 / 3.1410	[0 ∞]
FToRob	0.2099	[0 ∞]
LD*	0.09	{100, 200, 500}
LocToRob	0.25925	[0 ∞]
FT0y^T	0.156	[0 500] [4]
FT0z^T	0.156	[0 500] [5]
DyMod [6]	0.3768	[0 ∞]
Act*	25.01	[0 ∞]

1. Valeurs maximales obtenues sur 1000 exécutions de l'*Atome* sur la BeagleBoard
2. Il ne s'agit ici que de la réception des données envoyées par le PC de supervision qui est chargé de traiter les trames provenant du sonar, ici envoi d'un seul point d'impact à la fois.
3. Ces périodes de mise à jour dépendent du réglage du capteur (portée, nombre de points d'impact) et ont ici été arrondies au dixième de seconde.
4. Valeur obtenue par simulation.
5. Valeur obtenue par simulation.

Il est important de noter que les valeurs de T_{phy} relatives à la stabilité sont portées par les *Atomes* appartenant au *Domaine de Connaissance Task* qui sont indiqués par un ^{T} dans le tableau.

Tableau 10.2 – Valeurs des différents temps de transition au sein des *Blocs*

Bloc	t_{comm} [7] (ms)
S1	17.18 / 1.55
S2	0.1246
P1	0
P2	0
S3	0.1128
S4	0.795

Nous allons ici réutiliser les formules de diffusion présentées à l'Exemple 6.4. Toutes les durées et périodes seront exprimées en millisecondes. Commençons par calculer $t_{comp_{max}}(S_2)$ et $T_{exe}(S_2)$ respectivement à l'aide des équations (6.16) et (6.17) :

$$t_{comp_{max}}(S_2) = 0.47385 \tag{10.1}$$

$$T_{exe}(S_2) = \{100, 200, 500\} \tag{10.2}$$

Calculons ensuite $t_{comp_{max}}(S_1)$ et $T_{exe}(S_1)$ respectivement à l'aide des équations (6.14) et (6.15), en supposant l'utilisation du centrage en 3 dimensions (100 points d'impacts) :

$$t_{comp_{max}}(S_1) = 645.0197 \tag{10.3}$$

$$T_{exe}(S_1) = \{3770,\ 4020,\ 4500,\ 5650,\ 7900,\ 10200\} \tag{10.4}$$

De par ces deux calculs, nous pouvons maintenant déterminer $t_{comp_{max}}(P_1)$ et $T_{exe}(P_1)$. Pour le calcul de la première *propriété*, nous devons spécifier f_{P_1} dans (6.12), nous utilisons une fonction de diffusion série (voir par exemple (6.11)). Pour la seconde, nous utilisons (6.13) et nous obtenons ainsi :

6. Les temps sont indiqués pour l'*Atome* réellement utilisé dans nos applications (voir Section 9.1.3) et pas pour l'*Atome* simplifié considéré jusqu'alors pour l'évitement de parois.

7. Ce temps est calculé à partir de la pire durée d'échange de données entre les différentes entités (*Blocs* ou *Atomes*). La durée des opérations effectuées par le *Middleware* ContrACT (changements de contextes, messages entre modules et ordonnanceur) qui est très difficile à estimer précisément sera ignorée dans notre cas.

$$t_{comp_{max}}(P_1) = t_{comp_{max}}(S_1) + t_{comp_{max}}(S_2) + t_{comm}(P_1) = 645.49355 \quad (10.5)$$

$$T_{exe}(P_1) = \emptyset \quad (10.6)$$

Ainsi, la fonctionnalité n'est pas implémentable sur notre cible technologique puisque nous ne pouvons pas trouver de période commune à nos différentes applications. Les équations (10.3) et (10.4) nous permettent de mieux cerner le problème :

- Par (10.4), nous pouvons constater que le sonar est trop lent pour pouvoir être utilisé avec notre asservissement. En effet, la période maximale permise par le Loch Doppler (à cause des couplages temporels) et surtout l'ensemble de périodes garantissant la stabilité $T_{phy}(FT0) = [0\ 500]$ sont bien inférieures aux périodes de fonctionnement du sonar profilométrique.
- Sur la BeagleBone, le centrage en 3 dimensions, utilisant 100 points de mesure, représente une charge calculatoire trop importante pour nous permettre de respecter nos *Contraintes* temporelles et ainsi garantir la stabilité.

Pour solutionner le premier problème, nous pouvons soit changer de sonar soit modifier notre *Composition*. Comme il n'est évidemment pas possible de trouver des sonars ayant des fréquences de scan aussi élevées avec les technologies actuelles, nous sommes contraints de repenser notre *Composition*. Pour cela, il va nous falloir introduire des *Atomes* permettant un **découplage temporel** entre la *Connaissance Externe* représentant le sonar profilométrique et le reste de notre *Composition*.

Le problème de temps de calcul va nous contraindre à nous limiter à un test du centrage en 2 dimensions sur la BeagleBone. En effet, cette version qui peut notamment être utilisée pour se centrer dans un canal ne nécessite que l'utilisation de deux points sur l'axe $\vec{y}_B$, un à gauche et un à droite du robot. Cela permet donc de grandement soulager la charge calculatoire imposée à la BeagleBone.

10.2 Modification de la Composition pour permettre son implémentation

Il nous faut donc appliquer ces modifications à notre *Composition* pour permettre son implémentation. En plus des modifications précédemment évoquées, nous allons affiner le fonctionnement du sonar. En effet, lors de la conception de la *Composition*, nous avons fait la supposition que le sonar transmettait tous les points d'impact d'un scan (i.e. les 100 points) de manière simultanée. Or, en réalité le sonar va transmettre les points d'impacts au fur et à

mesure de leur réception. Nous ajoutons toutefois l'hypothèse, qui apportera plus de souplesse dans notre conception, que le *driver* du capteur peut ne pas transmettre qu'un seul point de manière simultanée (comme c'est le cas de celui que nous utilisons) mais qu'il peut en fournir plusieurs.

Nous allons ainsi présenter les modifications apportées à notre *Composition*.

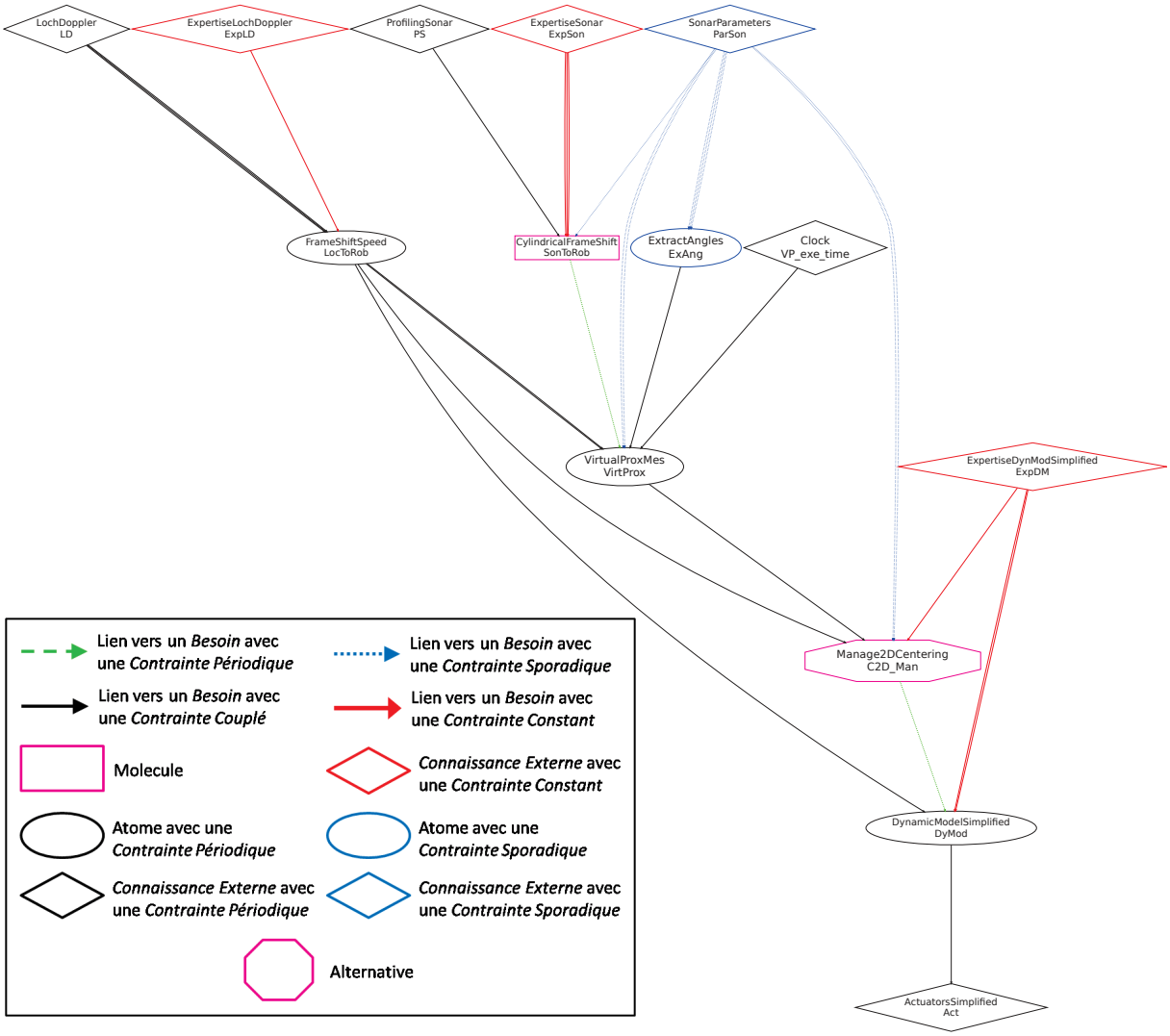

FIGURE 10.1 – Modifications apportées à la fonctionnalité d'évitement de parois

10.2.1 Modifications apportées à la Composition

La Figure 10.1 présente les modifications effectuées sur la fonctionnalité d'évitement de paroi.

Tout d'abord, comme le centrage est désormais effectué en 2 dimensions, nous avons retiré tous les liens relatifs à l'utilisation de la vitesse de montée/descente du robot w ainsi qu'à

l'accélération associée $\dot{w}$ et ceci afin d'alléger autant que possible la figure et les explications. En outre les *Instances* $FT0z$ et $Damp_z$ ne seront plus utilisées et l'évitement de parois sur l'axe $\vec{z}_B$ est remplacé par une autre fonctionnalité (un asservissement en profondeur) non représentée ici.

Nous devons préciser que, si dans nos explications nous n'utiliserons que deux points sonar, dans le *driver* du sonar, nous fusionnons plusieurs points autour des points situés sur l'axe $\vec{y}_B$ afin d'assurer plus de précision dans la mesure.

Nous avons en outre complété les paramètres du sonar. Outre sa période, sa portée et le nombre d'impacts dans le scan, nous ajoutons Ntr, le nombre de points d'impacts transmis à chaque cycle par le sonar (il vaudra 1 dans notre cas), *angle_start* et *angle_end* qui représentent les angles de début et de fin du scan (notre sonar profilométrique peut être paramétré afin de ne pas effectuer un scan sur 360°).

Afin d'assurer le découplage temporel entre le sonar profilométrique et le reste de la *Composition*, nous introduisons l'*Atome* $VirtualProximeter$. Celui-ci a pour *Paramètres d'Interface* et *Interface* :

$$IntPar(VPX) = \{Ent \in Fr, axS1 \in Ax, axS2 \in Ax, ax \in Ax\} \tag{10.7}$$

$$\begin{aligned}
IS(VPX) &= Array < CylindricPoint < Ent >> meas_pre && (10.8)\\
&\quad Array < CylindricPoint < Ent >> measSon_pre && (10.9)\\
&\quad Duration < NOFRAME, NOAXIS > t_pre && (10.10)\\
&\quad UInteger < NOFRAME, NOAXIS > cpt_in && (10.11)\\
Ne(VPX) &= Array < CylindricPoint < Ent >> measSon && (10.12)\\
&\quad Duration < NOFRAME, NOAXIS > t && (10.13)\\
&\quad Speed < Ent, axS1 > speed1 && (10.14)\\
&\quad Speed < Ent, axS2 > speed2 && (10.15)\\
&\quad UInteger < NOFRAME, NOAXIS > NR && (10.16)\\
&\quad UInteger < NOFRAME, NOAXIS > Ntr && (10.17)\\
&\quad Array < Angle < Ent, ax >> angles && (10.18)\\
Pr(VPX) &= Array < CylindricPoint < Ent >> meas && (10.19)
\end{aligned}$$

Le *Besoin measSon* correspond aux données transmises par le sonar profilométrique. Il s'agit d'un tableau de taille Ntr. Le *Besoin* t est la date d'exécution de l'*Atome* qui est

récupérée en faisant appel à la fonction dédiée de l'OS utilisé et est représentée par une *Connaissance Externe*, *Clock* dans notre graphe. Les *Besoins speed*1 et *speed*2 correspondent aux vitesses du robot (v et w) dans notre cas. Enfin, *angles* représente la liste des orientations des points d'impact. Ce *Besoin* est valué par l'*Atome ExtractAngles* qui est chargé de le déterminer à partir de NR, $angle\_start$ et $angle\_end$ du sonar.

La *Physique* de $VirtualProximeter$ est détaillée dans l'Annexe E mais nous devons toutefois souligner certains points essentiels. Tout d'abord nous supposons qu'entre deux mesures consécutives sur chaque rayon, seuls les déplacements du robot font varier la distance aux différents points d'impact mais que l'environnement lui est constant entre les deux mesures. C'est évidemment une hypothèse très contraignante mais la seule utilisable en l'absence de connaissance plus complète sur l'environnement entre les deux zones de mesure (carte, modèle géométrique). De fait, pour respecter cette hypothèse, il faut s'assurer que la vitesse d'avance du robot soit suffisamment faible. Notons $Tsonar$ la période de mise à jour du sonar. Nous allons noter d_{ctt} la distance maximale entre deux mesures pour laquelle l'hypothèse de constance peut être considérée comme valide et d_{scan} la précision souhaitée pour la reconstruction de l'environnement (distance maximale entre deux mesures là aussi). Il faut noter que ces deux valeurs proviennent généralement des spécialistes de l'environnement, la seconde traduisant souvent leur besoin (i.e. précision du modèle reconstruit pour être exploité dans leurs travaux) et la première servant de point d'entrée à leur expertise. Ainsi, la vitesse d'avance maximale du robot u_{max} peut être définie comme :

$$u_{max} = \frac{min(d_{ctt}, d_{scan})}{Tsonar} \tag{10.20}$$

Deuxièmement, cet *Atome* va renvoyer une mesure invalide tant que tous les points n'auront pas été mis à jour. Pour indiquer qu'un des impacts est invalide, nous avons décidé de fixer la distance qu'il mesure à 0 (ce qui est évidemment une distance que ne peut mesurer le capteur). De fait tant que nous n'aurons pas reçu au moins un impact sur chaque rayon lancé entre $angle\_start$ et $angle\_end$, nous fixerons à 0 l'ensemble des mesures. Bien que cela ne soit pas implémenté dans cette version de l'*Atome*, nous pouvons également considérer certains rayons invalides si nous constatons un problème (mesure incohérente, perte du Loch Doppler par exemple).

Dès lors, nous utilisons l'*Alternative* $Manage2DCentering$ afin de s'assurer que la fonctionnalité de centrage ne sera utilisée que si la mesure produite est valide. Cette *Alternative* est décrite Figure 10.2.

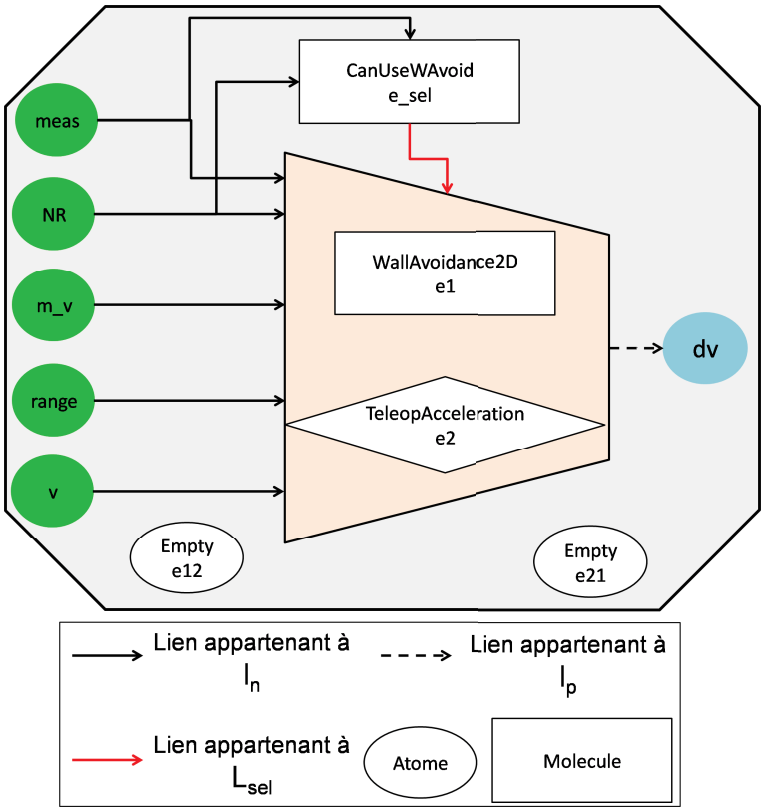

FIGURE 10.2 – *Alternative* permettant d'activer ou non l'évitement de parois en fonction de la validité des mesures sonar

La *Molécule WallAvoidance2D* regroupe toutes les *Entités Composables* mettant en œuvre notre fonctionnalité (voir Figure 5.3). Si nous ne pouvons l'utiliser car la mesure est considérée comme invalide, alors le contrôle de ce degré de liberté est rendu à l'utilisateur via une téléopération directe. L'*Entité Composable* de sélection est une *Molécule* dont le *Graphe Moléculaire* est représenté Figure 10.3.

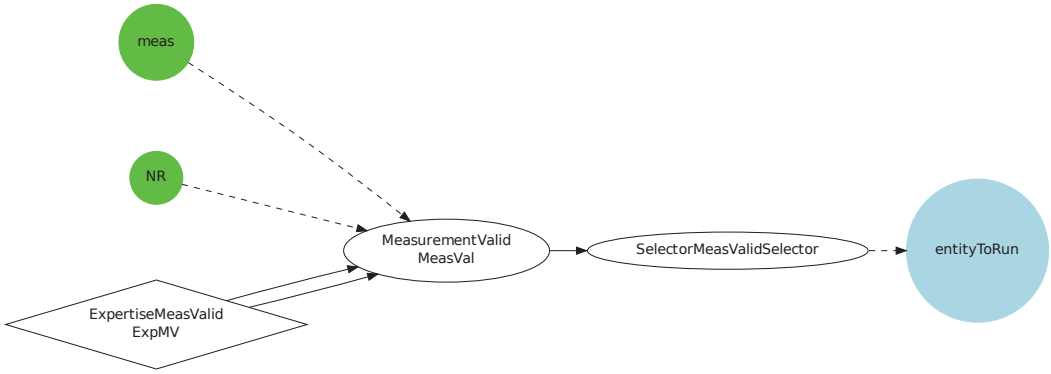

FIGURE 10.3 – *Entité Composable* de sélection de l'*Alternative Manage2DCentering*

Nous retrouvons un *Atome* chargé d'évaluer de manière périodique une propriété de notre système (ici la validité des mesures sonar) et un *Atome* de la famille des Sélecteurs chargé de déterminer l'*Entité Composable* à exécuter suivant la *Physique* :

$$
\begin{aligned}
&if(measValid == true)\\
&\{\\
&\quad entityToRun = "e1";\\
&\}\\
&else\\
&\{\\
&\quad entityToRun = "e2";\\
&\}
\end{aligned}
$$

L'*Atome* chargé de déterminer si la mesure est valide, $MeasurementValid$, va utiliser pour cela deux paramètres qui seront fixés par la *Connaissance Externe* $ExpertiseMeasurementValid$. Le premier $delta\_total$ sert à indiquer le seuil de points invalides dans l'ensemble du scan à partir duquel nous considèrerons une mesure invalide. Le second $delta\_consec$ sert à indiquer le seuil de points invalides consécutifs à partir duquel nous considèrerons une mesure invalide. Dans notre cas, nous avons choisi $delta\_total = delta\_consec = 1$, puisqu'avec seulement deux points utilisés, l'évitement de parois ne fonctionnera plus si un des points n'est pas valide.

La *Physique* de $MeasurementValid$ est définie comme :

$$
\begin{aligned}
&measValid = true;\\
&err\_ttl = 0;\\
&err\_cons = 0;\\
&for(i = 0; i < NR; i++)\\
&\{\\
&\quad if(meas[i].r == 0)\\
&\quad \{\\
&\quad\quad err\_ttl++;\\
&\quad\quad err\_cons++;\\
&\quad\quad if(err\_cons \geq delta\_consec)\\
&\quad\quad \{\\
&\quad\quad\quad measValid = false;\\
&\quad\quad\quad break;\\
&\quad\quad \}\\
&\quad \}\\
&\quad else\\
&\quad \{\\
&\quad\quad err\_cons = 0;\\
&\quad \}\\
&\}\\
&if(err\_ttl \geq delta\_total)\\
&\{\\
&\quad measValid = false;\\
&\}
\end{aligned}
$$

Nous allons maintenant reprendre notre étude des *Contraintes* et vérifier si notre *Composition* est implémentable ou non.

10.2.2 Nouvelle étude des Contraintes

Nous commençons notre phase d'étude des *Contraintes* en structurant le *Graphe d'Association de Connaissances* décrivant notre *Composition* (voir Chapitre 6). Comme nous utilisons une *Alternative*, nous obtenons deux *Graphes d'Association de Connaissances* temporellement indépendants. Le premier de ces deux graphes est présenté Figure 10.4 et utilise l'entité $e1$ de l'*Alternative*. Il est constitué de deux parties temporellement indépendantes (parties a et b de la figure), nous représentons ici leur décomposition en *Blocs*.

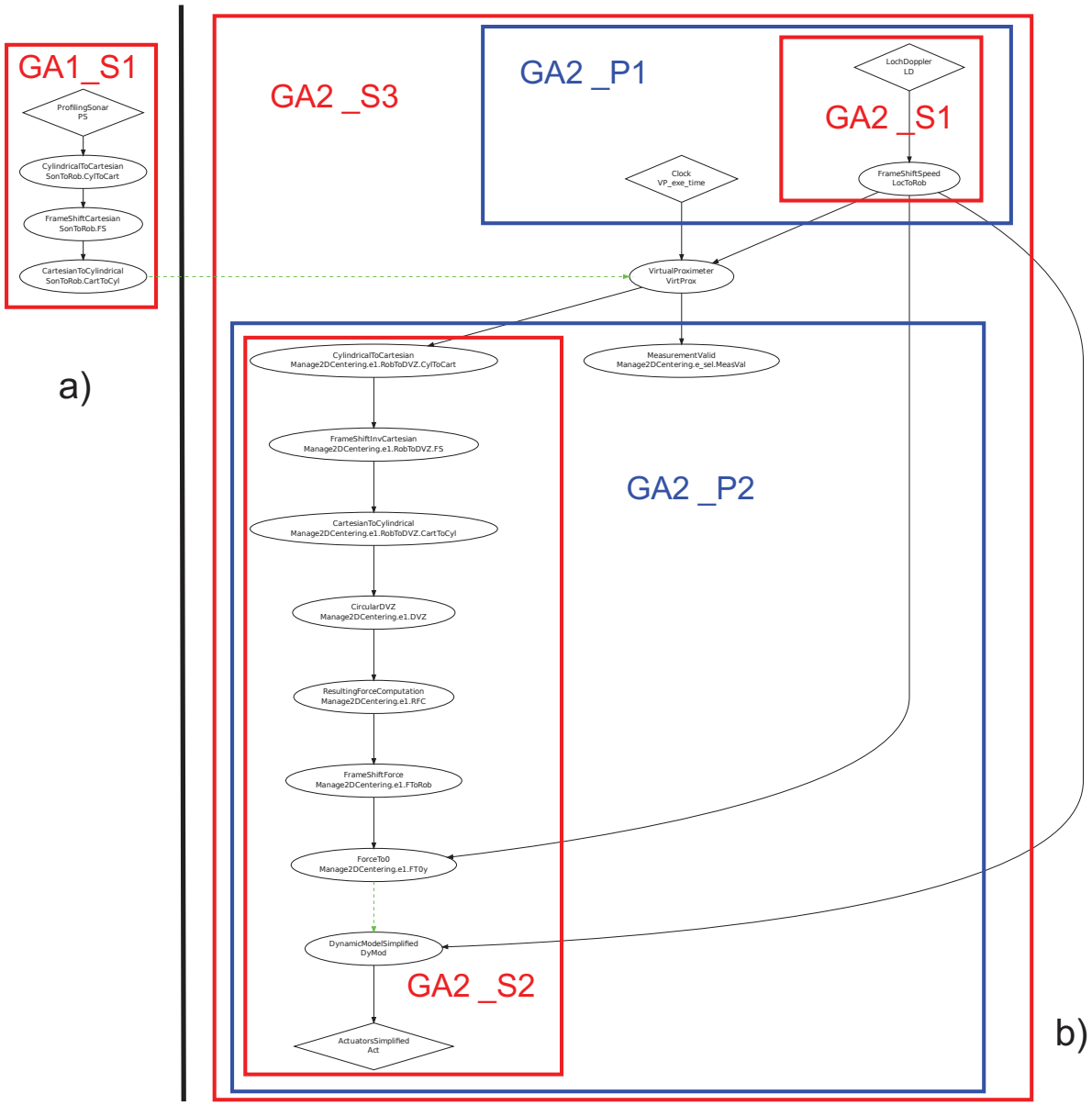

FIGURE 10.4 – Décomposition en *Blocs* du premier *Graphe d'Association de Connaissances* obtenu durant l'étude des *Contraintes*

Le second *Graphe d'Association de Connaissances* est présenté Figure 10.5 et utilise l'entité $e2$ de l'*Alternative*. Il est constitué de trois parties temporellement indépendantes (parties a, b et c de la figure), nous représentons ici leur décomposition en *Blocs*.

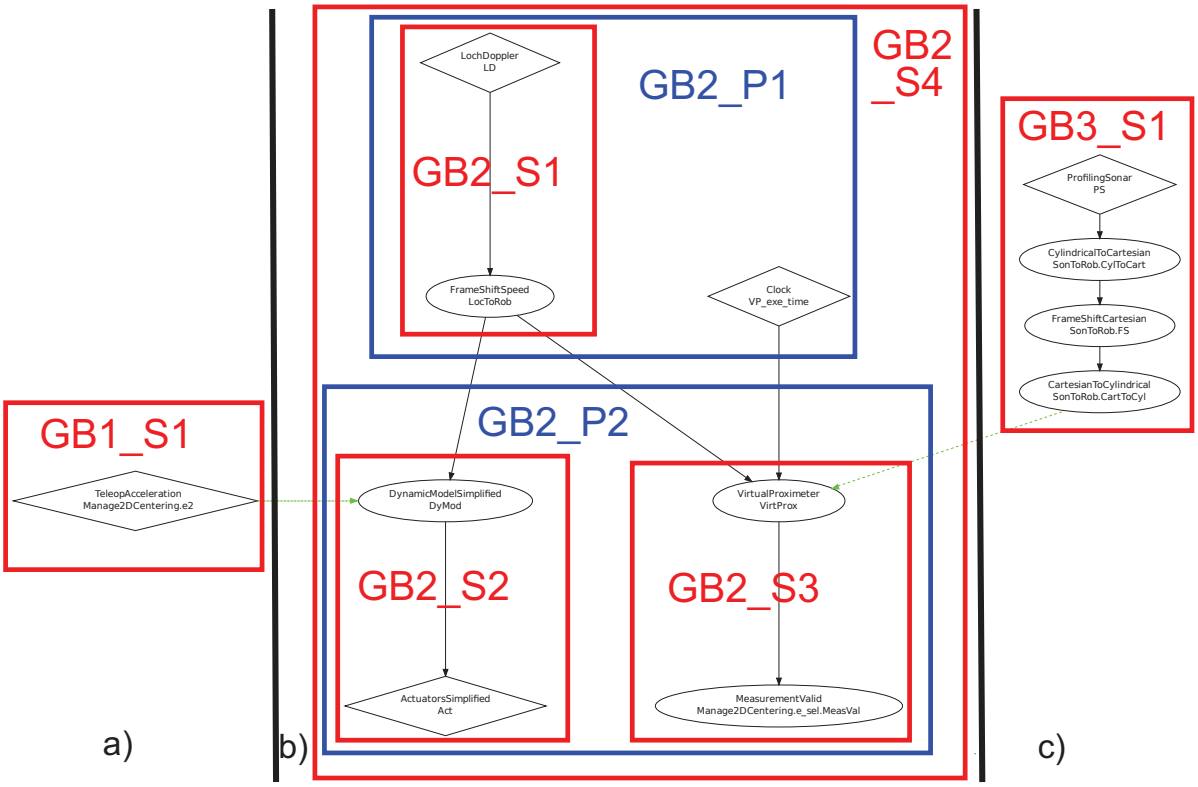

FIGURE 10.5 – Décomposition en *Blocs* du second *Graphe d'Association de Connaissances* obtenu durant l'étude des *Contraintes*

Nous allons maintenant appliquer à nouveau les formules de diffusion pour vérifier si notre *Composition* est implémentable. Nous nous focaliserons sur le premier des deux graphes (Figure 10.4). Nous rappelons également que, comme nous fonctionnons en monoprocesseur, les formules de diffusion pour les *Blocs* parallèles seront identiques à celles pour les *Blocs* série. Pour la première partie temporellement indépendante du *Graphe d'Association de Connaissances*, nous avons :

$$t_{comp_{max}}(GA1\_S1) = t_{comp_{max}}(PS) + t_{comp_{max}}(SonToRob.CylToCart) + t_{comp_{max}}(SonToRob.FS) + t_{comp_{max}}(SonToRob.CartToCyl) + t_{comm}(GA1\_S1) \quad (10.21)$$

$$T_{exe}(GA1\_S1) = [t_{comp_{max}}(GA1\_S1)\ \infty] \cap T_{phy}(PS) \cap T_{phy}(SonToRob.CylToCart) \cap T_{phy}(SonToRob.FS) \cap T_{phy}(SonToRob.CartToCyl) \cap T_{Need}(VirtProx.measSon) \quad (10.22)$$

et pour la seconde partie du graphe, nous avons

$$t_{comp_{max}}(GA2\_S3) = t_{comp_{max}}(GA2\_P1) + t_{comp_{max}}(GA2\_P2) + t_{comm}(GA2\_S3) \quad (10.23)$$

$$T_{exe}(GA2\_S3) \;=\; [t_{comp_{max}}(GA2\_S3) \;\; \infty] \cap T_{phy}(GA2\_P2) \cap T_{phy}(GA2\_P2) \quad (10.24)$$

Pour évaluer ces différentes expressions, nous allons utiliser les données numériques du Tableau 10.1 (données exprimées pour des mesures de deux points) et des Tableaux 10.3, 10.4 et 10.5. Il est important de noter que pour l'*Atome* $VirtualProximeter$, nous avons utilisé pour valeurs $NR = 2$ et $Ntr = 1$.

Tableau 10.3 – Valeurs des différentes *Contraintes* temporelles dans l'exemple d'évitement de parois modifié

Atome	$t_{comp_{max}}$ (ms)	T_{phy} (ms)
VirtProx	5.931	[0 ∞]
VP_exe_time	0.0375	[0 ∞]
Manage2DCentering.e_sel.MeasVal	2.258	[0 ∞]
Manage2DCentering.e2	0.06	[0 ∞]

Tableau 10.4 – Valeurs des *propriétés* des *Besoins* ayant des *Contraintes* temporelles de *nature Périodique*

Besoin	T_{Need} (ms)
VirtProx.measSon	[0 ∞]
DyMod.dv	[0 ∞]

Les résultats numériques ainsi obtenus sont proposés au Tableau 10.6. Les *Blocs* représentant des graphes temporellement indépendants sont notés en gras.

Enfin, pour finaliser la vérification de l'implémentabilité de notre *Composition*, nous devons vérifier que :

$$t_{comp_{max}}(Manage2DCentering.e12) \leq t_{exe_{max}}(e12) \quad (10.25)$$

$$t_{comp_{max}}(Manage2DCentering.e21) \leq t_{exe_{max}}(e21) \quad (10.26)$$

Tableau 10.5 – Valeurs des différents temps de transition au sein des *Blocs* dans l'exemple modifié, les *Blocs* non représentés ont un temps de transition nul

Bloc	t_{comm} (ms)
GA1_S1 / GB3_S1	0.2444
GA2_S1 / GB2_S1	0.1246
GA2_S2	2.1247
GA2_S3	1.3205
GB2_S2	0.1128
GB2_S3	0.6219
GB2_S4	0.3621

Tableau 10.6 – *Propriétés* temporelles des différents *Blocs* identifiés

Bloc	$t_{comp_{max}}$ (ms)	T_{exe} (ms)	*Bloc*	$t_{comp_{max}}$ (ms)	T_{exe} (ms)
GA2_S1	0.47385	$\{100, 200, 500\}$	GB2_S1	0.47385	$\{100, 200, 500\}$
GA2_P1	0.51135	$\{100, 200, 500\}$	GB2_P1	0.51135	$\{100, 200, 500\}$
GA2_S2	48.6815	[48.6815 500]	GB2_S2	25.4996	[25.4996 ∞]
GA2_P2	50.9395	[50.9395 500]	GB2_S3	8.8109	[8.8109 ∞]
GA2_S3	52.77135	$\{100, 200, 500\}$	GB2_P2	34.3105	[34.3105 ∞]
GA1_S1	$12,3412$	$\{2500,\ ...,\ 10200\}$	GB2_S4	35.18395	[35.18395 ∞]
GB1_S1	0.06	[0.06 ∞]	GB3_S1	$12,3412$	$\{2500,\ ...,\ 10200\}$

Nous avons choisi de fixer $t_{exe_{max}}(e21) = t_{exe_{max}}(e12) = 100\ ms$ (soit un maximum d'un cycle d'exécution pour effectuer la commutation). Avec le choix d'implémentation de l'*Alternative* effectué (voir section suivante), les durées $t_{comp_{max}}$ sont limitées à la fixation d'un paramètre de module par un superviseur. Si avec l'architecture actuelle nous ne pouvons mesurer ce temps avec précision, il demeure d'un ordre de grandeur inférieur à la milliseconde. Ainsi, les conditions (10.25) et (10.26) sont respectées. Notre *Composition* est donc **implémentable**.

Il nous faut maintenant choisir une période d'exécution pour chaque partie du graphe temporellement indépendante parmi celles possibles. Afin d'éviter de devoir réaliser des commutations de schéma en cas de commutation dans l'*Alternative* et ainsi regrouper un maximum de modules dans un même schéma pour limiter les problèmes de déterminisme qu'entraîne l'impossibilité de poser des contraintes de précédence entre schémas, nous avons choisi de faire fonctionner tous les *Atomes* à une période de 100 ms à l'exception de ceux couplés au sonar qui fonctionneront à une période de 2500 ms.

10.2.3 Exemple de projection sur le Middleware ContrACT

Nous avons ainsi déterminé que notre *Composition* est implémentable. Nous allons maintenant discuter de la répartition des différents *Atomes* au sein des modules ContrACT. Nous ne traiterons ici que du cas des modules périodiques évoqués dans les figures 10.4 et 10.5 en appliquant certaines des règles présentées dans le Chapitre 7.

Premièrement, en appliquant la Règle 5, nous pouvons déterminer la répartition d'*Atomes* dans des modules proposée Figure 10.6. Il faut néanmoins noter une exception à cette règle, il s'agit de la *Connaissance Externe* VP_exe_time. En effet, comme elle doit transmettre la date de début d'exécution de l'*Atome* $VirtProx$, elle doit se trouver dans le même module que celui-ci.

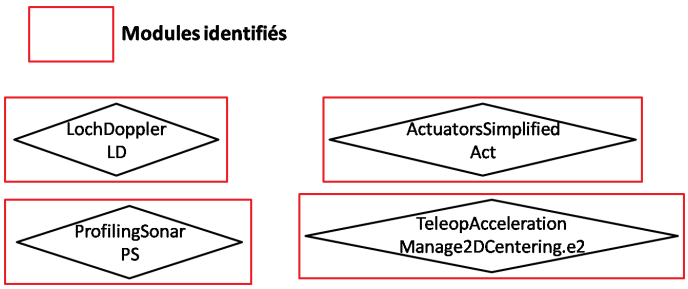

FIGURE 10.6 – Répartition d'*Entités Composables* dans des modules identifiés par la Règle 5

L'application de la Règle 6 (ainsi que de la Règle 9) nous indique que $DyMod$ et $Manage2DCentering.e2.FT0y$ ne peuvent pas faire partie du même module. Par voie de conséquence, étant donné que les autres modules sont tous situés en amont de $Manage2DCentering.e2.FT0y$, nous devons isoler $DyMod$ dans un module dédié (Règle 7).

Similairement l'application de la Règle 6 nous fait regrouper les *Atomes* $SontToRob.CylToCart$, $SontToRob.FS$ et $SontToRob.CartToCyl$ dans un même module.

Par la Règle 9, nous savons que les différentes *Entités Composables* appartenant à l'*Alternative* doivent être situés dans des modules indépendants des autres. En outre au sein de l'*Alternative*, l'application de la Règle 6 nous impose d'isoler l'*Atome* $Manage2DCentering.e\_sel.MeasVal$ dans un module dédié. La répartition des *Atomes* appartenant à l'*Alternative* est rappelée Figure 10.7. Cette *Alternative* sera mise en œuvre comme une *Alternative partielle* (Règle 12).

Enfin, les trois *Atomes* restant, VP_exe_time, $VirtProx$ et $LocToRob$ peuvent appartenir à un même module.

Nous avons appliqué toutes les règles permettant de définir la répartition des *Atomes* entre les modules. Maintenant, le concepteur a la liberté de réaliser un découplage plus fin s'il le souhaite (notamment pour des raisons de modularité). Dans ce cas, nous avons choisi de

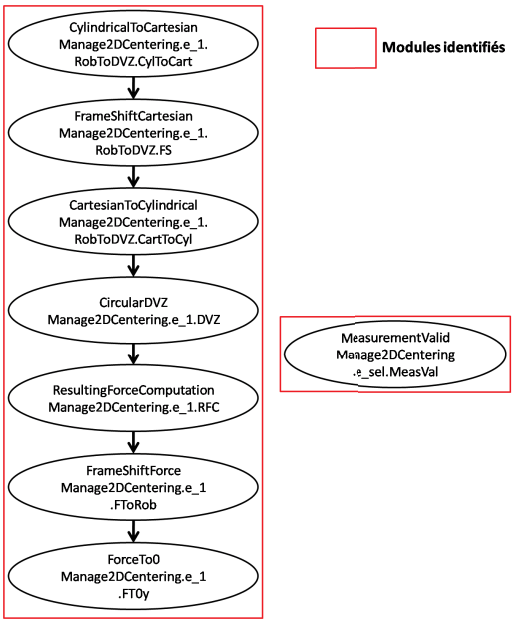

FIGURE 10.7 – Répartition d'*Entités Composables* dans des modules identifiés par la Règle 9

séparer $LocToRob$ de VP_exe_time et $VirtProx$ car il s'agit d'une fonctionnalité susceptible d'être réutilisée dans un cadre plus large que l'évitement de parois. Similairement, nous avons subdivisé celle-ci en trois modules comme montré Figure 10.8.

Considérons ensuite la répartition de ces modules en schémas et le choix de la période pour chacun. Pour cela, par la Règle 8, nous pouvons ainsi regrouper les modules en deux schémas présentés Figure 10.8 et Figure 10.9. Par la Règle 17, le premier de ces deux schémas aura une période de 100 ms et le second une période de 2500 ms. Enfin, les liaisons entre ces différents modules seront assurées par des flux de données (Règle 14).

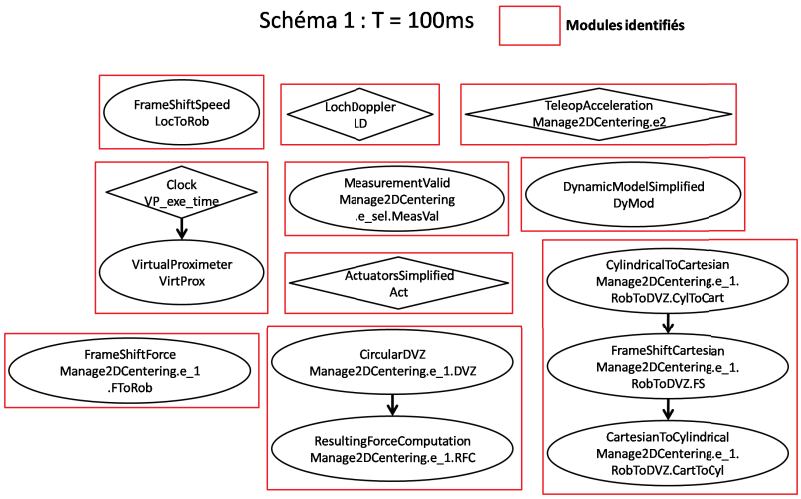

FIGURE 10.8 – Le premier des deux schémas implémentant notre *Composition*

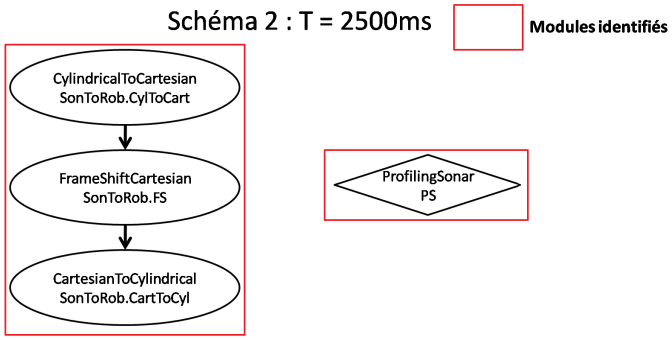

FIGURE 10.9 – Le second des deux schémas implémentant notre *Composition*

10.3 Illustration par la simulation

10.3.1 Un environnement simple

Pour illustrer notre *Composition* nous allons commencer par tester notre asservissement en simulation dans un environnement régulier pour vérifier sa bonne convergence. Dans cette simulation, nous avons $Tsonar = 2.5\ s$. Ici, $Tsonar$ représente la durée pour effectuer un scan avec une ouverture de 180°. Le bruit maximal sur la mesure sonar est $Sonar_{noise} = 0.1\ m$. Le bruit sur les mesures de vitesse est $v_{noise} = 0.01\ m.s^{-1}$. Le robot avance à une vitesse de $0.5\ m * s^{-1}$ et $K_{DVZ} = 7$.

L'environnement utilisé est présenté Figure 10.10. Dans cette figure, la trajectoire suivie par le robot est représentée en noir.

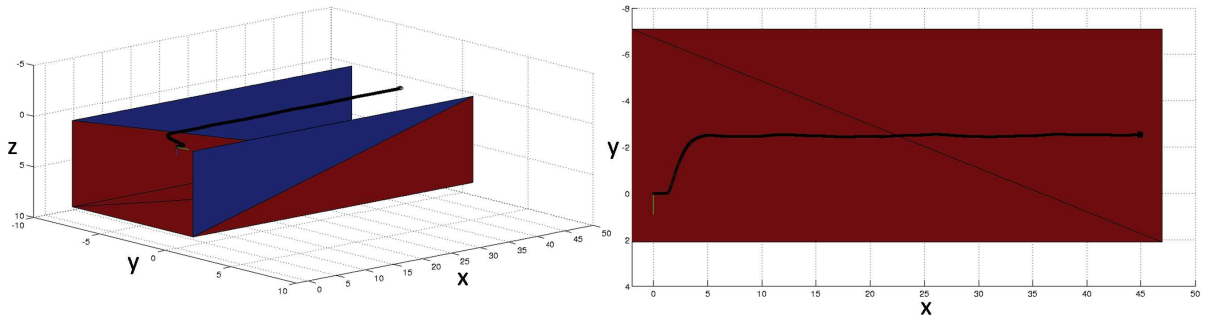

FIGURE 10.10 – Premier environnement de test et trajectoire suivie par le robot

La Figure 10.11 représente la position y du robot par rapport à la valeur cible qui correspond au centre de l'environnement. Nous pouvons constater que notre robot se centre à l'incertitude sur les mesures près, confirmant la bonne convergence du contrôleur. De plus en début d'exécution (nous pouvons également le constater dans les figures suivantes), le robot ne réagit pas avant 2.5 secondes soit la période choisie pour le sonar. En effet, tant que nous n'avons pas reçu une mesure par rayon, l'asservissement ne peut fonctionner et grâce à l'*Alternative* la fonctionnalité n'est activée qu'une fois toutes les mesures attendues reçues.

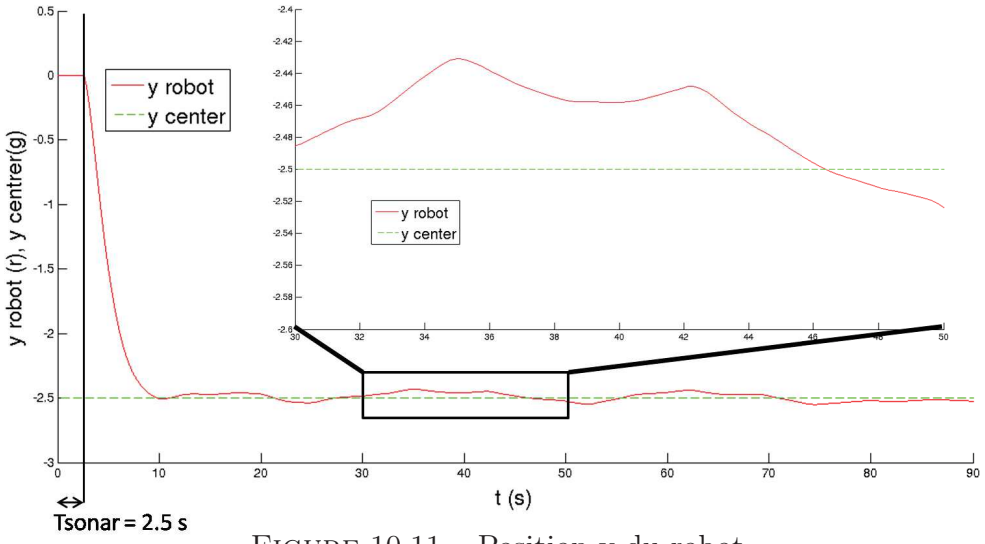

FIGURE 10.11 – Position y du robot

La Figure 10.12 représente la vitesse de glissement du robot, v. La courbe rouge représente la vitesse réelle du robot et la bleue la vitesse bruitée fournie par le Loch Doppler. Nous voyons que si la vitesse est faible, elle ne s'annule pas à cause du bruit sur les capteurs.

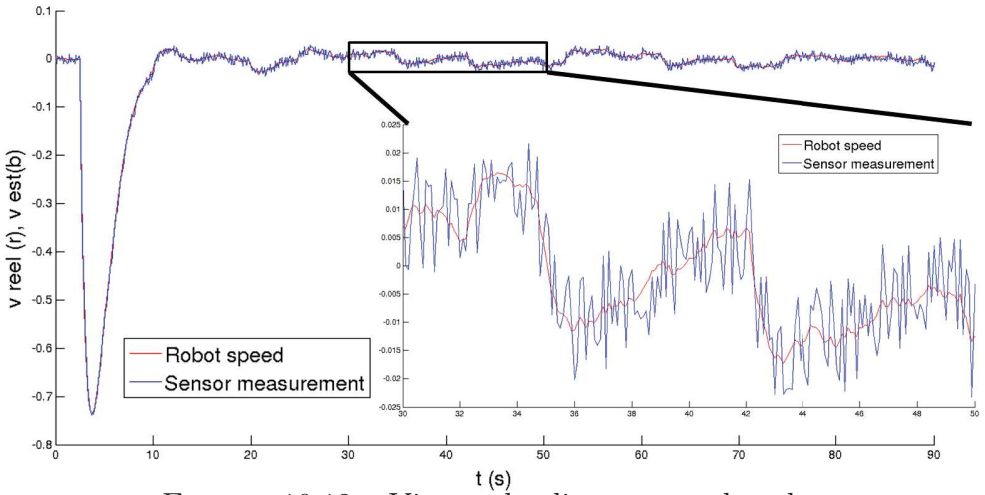

FIGURE 10.12 – Vitesse de glissement v du robot

Enfin, la Figure 10.13 présente les consignes d'accélération obtenues en sortie de l'*Atome* $ForceTo0$.

10.3.2 Un environnement de type canal

Nous terminons par une simulation dans l'environnement présenté Figure 10.14. Les paramètres des capteurs et du contrôleur sont les mêmes que pour la simulation précédente.

Les figures suivantes présentent les différents résultats de simulation. La Figure 10.17 présente la trajectoire suivie par le robot dans le canal. Le robot reste ainsi dans la partie centrale

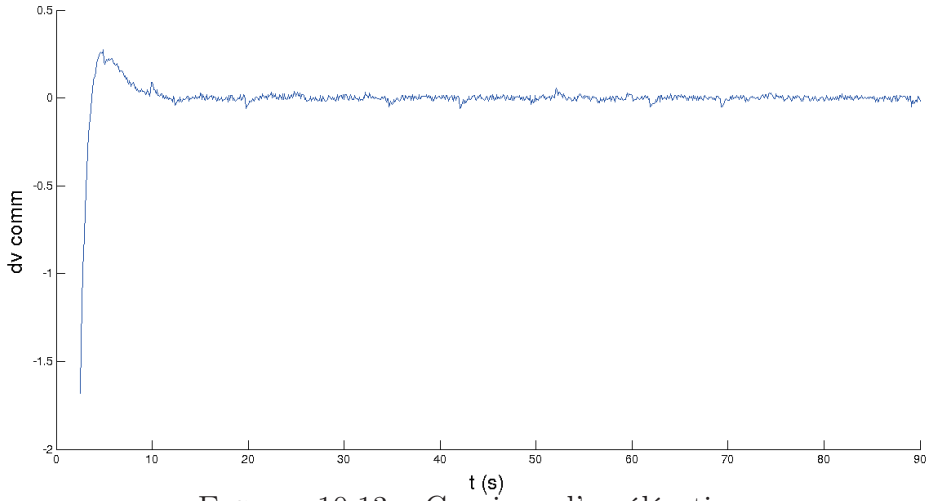

FIGURE 10.13 – Consigne d'accélération

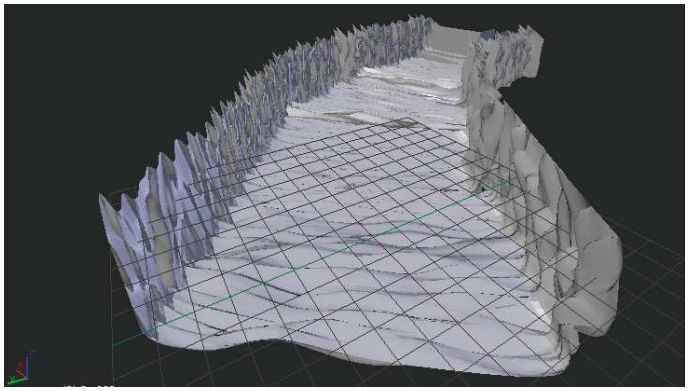

FIGURE 10.14 – Second environnement de simulation

du canal.

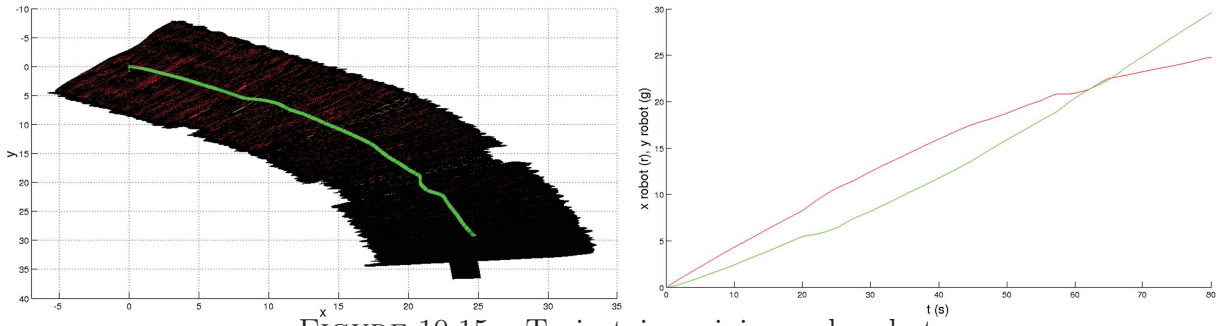

FIGURE 10.15 – Trajectoire suivie par le robot

Enfin les figures 10.16 à 10.18 présentent la vitesse de glissement du robot, la consigne d'accélération produite par l'asservissement et le cap suivi par le robot. Les deux premières ne s'annulent pas à cause du bruit capteur et de l'orientation du robot qui ne suit jamais exactement le cap théorique ce qui oblige le centrage à fournir plus d'efforts pour empêcher le robot de dériver du centre du canal. Nous pouvons enfin noter qu'à nouveau l'évitement de parois n'entre en action qu'une fois un premier scan effectué par le sonar.

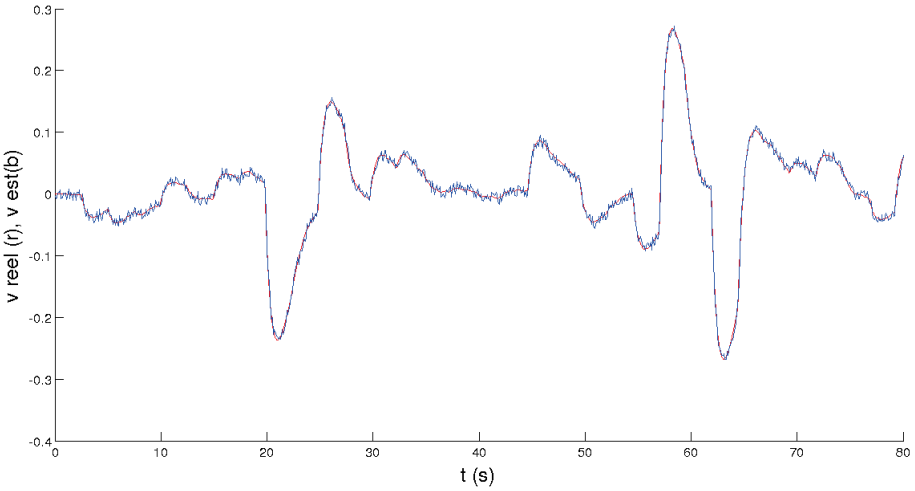

FIGURE 10.16 – Vitesse de glissement v du robot

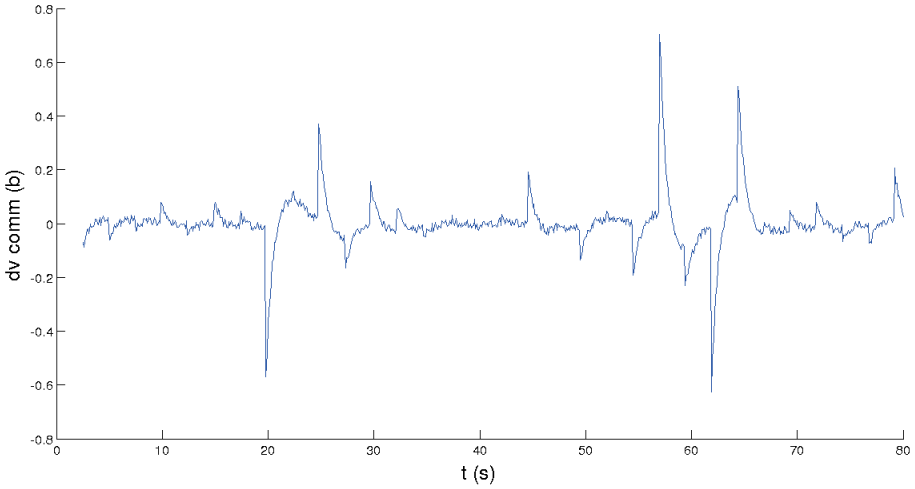

FIGURE 10.17 – Consignes d'accélération calculées par l'évitement de parois

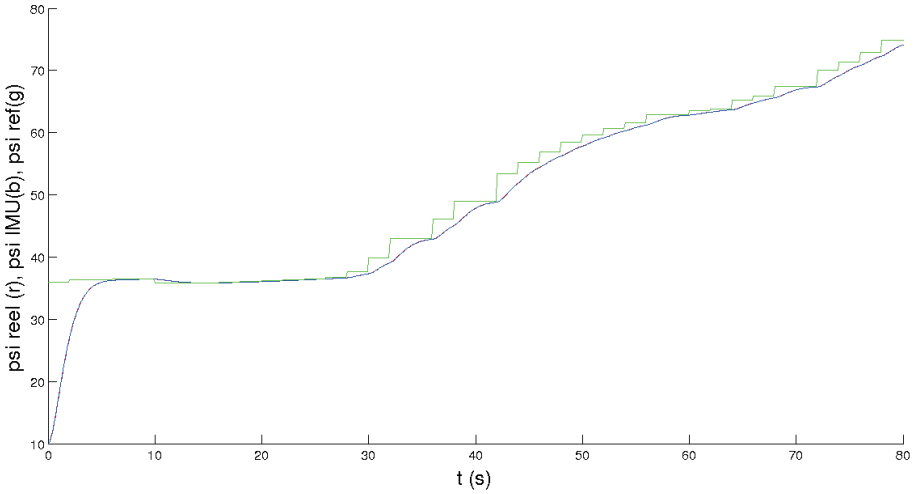

FIGURE 10.18 – Cap suivi par le robot

10.4 Points clés du chapitre

- La phase d'étude des *Contraintes* nous a permis de mettre en lumière l'impossibilité d'implémenter notre loi de commande exemple à cause de certains couplages temporels.
- Notre approche nous a permis d'identifier le problème et les modifications à apporter à notre *Composition*.
- Notre structuration, qui identifie clairement la contribution de chaque entité, nous a permis d'appliquer ces modifications afin de pouvoir implémenter la *Composition* tout en s'assurant de respecter les *Contraintes* temporelles liées à la stabilité.

Conclusion

Nos travaux portent sur la robotique sous-marine pour l'exploration d'environnements confinés ou faible fond. Notre objectif est l'exploration d'aquifères karstiques. Des environnements aussi complexes nécessitent de mettre en place les fonctionnalités robotiques offrant au robot un degré d'autonomie suffisant pour pouvoir exécuter sa mission en toute sécurité. Pour cela, il est nécessaire d'intégrer des connaissances provenant des différents spécialistes des environnements dans les stratégies de commande. Celles-ci sont amenées à évoluer au fil des missions qui vont apporter de nouvelles données sur les environnements. Les approches traditionnelles de développement de lois de commande, basées sur une conception monolithique, ne sont dès lors plus adaptées aux besoins d'expressivité et d'évolutivité.

Nous proposons donc un nouveau formalisme de description des architectures de contrôle centré sur les connaissances. Celui-ci comprend une description modulaire mettant en relief les connaissances, leurs contributions et les interactions entre elles. Cette approche d'ingénierie dirigée par les modèles se base sur un certain nombre d'*Entités Composables* qui servent à encapsuler les connaissances et sont assemblées pour décrire des *Compositions* qui sont notre représentation du contrôleur du robot. Ces *Entités Composables* sont structurées autour de deux éléments essentiels, une *Physique* qui contient les connaissances et une *Interface* permettant l'accès aux connaissances encapsulées dans l'entité. Les *Eléments d'Interface* contiennent toutes les informations nécessaires à l'utilisation de l'entité sans avoir une maîtrise précise de son contenu. Ils sont ainsi caractérisés par leur *Type*, le repère dans lequel est exprimée la donnée et l'axe du repère selon lequel elle évolue.

Différentes sortes d'*Entités Composables* sont définies en fonction de leur apport dans la définition des *Compositions*. Premièrement, les *Atomes* sont les entités minimales et indivisibles qui sont à la base de notre approche. En plus d'encapsuler des connaissances, ils portent les informations nécessaires à l'implémentation de notre architecture de contrôle sur une cible technologique. Ces *Contraintes* servent à prendre en compte à la fois les besoins des automaticiens pour garantir la stabilité des stratégies de contrôle mises en œuvre et les propriétés de la cible d'implémentation. Les *Molécules* sont le second type d'entité. Il s'agit d'*Entités Composables* permettant d'assembler des *Entités Composables* afin de mieux structurer nos

Compositions et de faciliter leur réutilisation. Enfin, les *Alternatives* servent à décrire comment adapter des connaissances en fonction de la situation rencontrée par le robot. Elles comprennent les différentes *Entités Composables* substituables selon les situations, la description des conditions guidant le choix de l'entité à exécuter ainsi que les actions à entreprendre lorsque l'*Entité Composable* à exécuter change.

A ce formalisme de représentation, nous associons une méthodologie qui va nous permettre de guider l'implémentation de notre architecture de contrôle sous forme de logiciel de contrôle. Une première phase consiste à manipuler notre *Composition*, qui peut être représentée sous forme de *Graphe d'Association de Connaissances*, afin d'étudier les *Contraintes* qui portent dessus. Ce processus se déroule en plusieurs étapes. Tout d'abord, nous remplaçons les *Molécules* et *Alternatives* par les *Atomes* qu'elles contiennent (dépliage) et sur lesquels reposent les *Contraintes*. Nous pouvons ensuite simplifier la *Composition* en enlevant les *Liens* considérés comme identiques et en se concentrant uniquement sur les entités s'exécutant sur une base périodique. Nous pouvons ensuite identifier les différentes parties de la *Composition* qui sont indépendantes temporellement. Enfin, pour chacune d'entre elles, nous fusionnons les *Contraintes* portant sur chaque *Atome* afin d'en déduire les périodes d'exécution possibles et de choisir celle qui sera effectivement utilisée. A partir de là, une phase de projection est chargée de guider le développeur dans la structuration du logiciel de contrôle. S'appuyant sur un nombre de règles spécifiques au *Middleware* utilisé, elle indique quels mécanismes utiliser et quelles erreurs éviter pour permettre l'implémentation de la loi de commande en respectant les *Contraintes* fixées. Nous avons illustré cette phase sur l'exemple concret du *Middleware* temps-réel ContrACT.

Enfin, nous avons validé notre approche pour la mener jusqu'aux expérimentations. Cette validation s'est faite suivant une approche progressive qui comprend différentes étapes accompagnant les grands jalons de notre méthodologie. La première est la simulation fonctionnelle qui vise à vérifier que la loi de commande a bien le comportement attendu. Les tests unitaires et d'intégration permettent ensuite de s'assurer que les entités logicielles implémentant nos *Atomes* ont le fonctionnement attendu. La simulation *Hardware-in-the-Loop* est la dernière étape avant les expérimentations. Elle permet de vérifier le bon fonctionnement du logiciel de contrôle. Nous avons enfin testé différentes lois de commande développées avec notre approche sur notre vecteur robotique, le ROV Jack, à la fois en piscine et en environnement naturel. Celui-ci, disponible en plusieurs versions suivant les besoins de l'application, a permis de mettre en évidence l'apport de notre approche en termes d'évolutivité. Mais la validation de notre approche et la mise en lumière de son apport ne pourront être effectives qu'une fois celle-ci confrontée à des contextes applicatifs divers au travers de diverses expérimentations

dans des milieux variés, avec l'exploration d'aquifères karstiques comme objectif.

Ainsi, nos travaux constituent une base de travail qui ouvre sur de nombreuses perspectives.

L'une des premières est l'amélioration de l'outil présenté dans l'Annexe D. Il est nécessaire à la fois d'améliorer les fonctionnalités existantes et de lui en ajouter de nouvelles afin d'automatiser un maximum d'étapes du processus et de rendre son utilisation toujours plus simple et intuitive pour les utilisateurs. Maintenant que la sémantique et les principaux concepts de notre approche ont été clairement posés, il s'agit principalement de problématiques techniques qui peuvent être traitées à court ou moyen terme. On peut ainsi évoquer les points suivants :

- Développement d'un langage dédié : comme nous l'avons expliqué dans l'Annexe D, nous avons utilisé le langage XML pour la description de nos *Entités Composables* et *Compositions*. Néanmoins, la syntaxe résultante s'avère assez lourde et relativement peu intuitive. Le développement d'un langage plus spécifique s'avère donc pertinent afin de gagner en simplicité dans la description des entités. De plus, celle-ci pourrait plus facilement intégrer une description générique de la *Physique*. Mais cela impose de développer un *parser* et un vérificateur syntaxique spécifiques à ce langage.
- Génération automatique de documentation à partir de la description d'une *Entité Composable* : documenter nos entités est indispensable afin d'en permettre une utilisation aisée par le plus grand nombre et tracer les différentes modifications effectuées tout au long de leur utilisation. Là encore, la structure très normalisée de nos entités offre la possibilité d'une automatisation du processus de documentation via la génération automatique d'un fichier LaTeX ou HTML par exemple. Cela nécessite également un enrichissement de la description de nos entités avec l'ajout d'informations spécifiques à la documentation (auteur, date, numéro de version, description, historique des modifications ...).
- Automatisation de la phase de test unitaire : comme nous l'avons présenté Section 8.2, le test unitaire permet de valider individuellement l'implémentation de chaque *Atome* et, sur la cible d'implémentation, permet d'évaluer son temps maximal d'exécution. Pour simplifier le travail des utilisateurs, il serait intéressant d'automatiser cette phase. Il faudrait tout d'abord définir une structuration de la description des vecteurs de test (en XML par exemple). Dès lors, il serait possible de générer une application permettant de tester l'entité logicielle que l'utilisateur n'aurait alors plus qu'à compiler et exécuter sur la cible d'implémentation. Une extension de cette approche aux tests d'intégration serait également intéressante.

- Intégration de l'ensemble des fonctionnalités : pour l'instant, notre outil logiciel n'intègre que les fonctionnalités de définition des *Entités Composables* et de génération de code des *Atomes* et *Molécules simples*. La représentation des *Compositions* sous forme de *Graphe d'Association de Connaissances* et la phase d'étude des *Contraintes* sont pour l'instant gérées à part. Or, pour notamment permettre d'utiliser pleinement la vérification des *Compositions* (i.e. vérifier que l'*Instanciation* des différentes entités respecte bien la définition qui en a été faite), il est nécessaire d'intégrer le traitement des *Graphes d'Association de Connaissances* dans notre logiciel.
- Ajout de génération vers d'autres cibles : nous n'effectuons, pour l'instant, que de la génération de code de nos *Atomes* sur l'API que nous avons développée. Il serait intéressant, suivant les besoins des utilisateurs, d'intégrer la génération de code vers d'autres API, permettant ainsi une couverture plus large des besoins utilisateurs.
- Interface de conception graphique : pour rendre plus intuitive la définition des *Compositions*, *Molécules* et *Alternatives*, il serait pertinent de proposer une interface de conception graphique (similaire par exemple à ce que proposent Simulink ou RobotFlow [CLM+04]). Mais il faudrait faire attention à la définition graphique des *Compositions* car nous avons vu que celles-ci, même simples, contiennent un nombre important d'*Entités Composables*. Dès lors, leur représentation graphique risque de devenir très rapidement illisible et donc inexploitable.

Un second point nécessitant plus de travaux concerne la mise en œuvre de notre approche. En effet, comme souligné aux Chapitres 7, 9 et 10, certaines limitations du *Middleware* ContrACT nous ont forcé à implémenter certains mécanismes, notamment ceux liés aux *Alternatives*, d'une manière dégradée par rapport à ce qui était initialement envisagé. Dès lors, des modifications structurelles de ContrACT ou l'utilisation d'un autre *Middleware* nous permettraient de mettre en œuvre certains aspects de notre approche d'une manière plus efficiente.

Une première modification à faire serait de permettre soit de poser des contraintes de précédence entre schémas de même période, soit de pouvoir commuter (activer et désactiver) en ligne des modules au sein d'un schéma. Nous pourrions ainsi réaliser des changements ciblés de modules au sein de notre architecture de contrôle afin de mettre en œuvre les *Alternatives* via un mécanisme de supervision comme défini initialement. Il s'agit de modifications importantes qui nécessitent à la fois une réflexion théorique, afin de repenser le fonctionnement de l'ordonnanceur de ContrACT, et la résolution des questions techniques au niveau du *Middleware* afin

d'implémenter ces modifications. L'ensemble de ces travaux sont réalisables à moyen terme.

Un autre changement concernerait la possibilité de changer en ligne les périodes d'exécution de nos schémas (alors que ce n'est actuellement possible que par un changement de schéma, qui n'est donc pas une solution très efficace). Cela permettrait d'amorcer en parallèle une réflexion théorique à plus long terme sur l'adaptation en ligne des périodes. Celle-ci a un intérêt à la fois pour des problématiques liées à la stabilité du contrôle (comme souligné Section 2.1.1, de nombreux facteurs influent sur la stabilité dont certains ne peuvent être pleinement pris en compte lors de l'implémentation) et à certains travaux liés à l'*autonomic computing*.

L'*autonomic computing* est un domaine qui vise à décrire des méthodes pour adapter l'exécution d'un processus informatique ou le fonctionnement d'un processeur pour s'adapter aux conditions d'exécution (pannes, charge calculatoire). Dans le contexte de la robotique d'intervention, où les ressources calculatoires et énergétiques sont limitées, ces travaux ont commencé à intéresser le domaine de la robotique afin d'améliorer l'exploitation de ces ressources. Notamment, il est possible d'adapter la période d'exécution d'une loi de commande afin de répondre par exemple à une surcharge calculatoire. Dans ce cas, nos travaux peuvent permettre d'explicitement représenter les vérifications et modifications réalisées par les algorithmes mis en œuvre dans le cadre de ces approches en utilisant les *Alternatives*. Dans le cas où ces algorithmes visent à réguler l'ordonnancement [SSS14], il est également possible de les décrire en explicitant l'interface entre les *Atomes* qui contiendraient les algorithmes d'adaptation et l'ordonnanceur lui-même, en utilisant une *Connaissance Externe*.

Qui plus est, en intégrant ces éléments à notre approche, il est possible d'évaluer leur impact temporel sur l'exécution du système. Notre approche d'étude des *Contraintes* nous permet également d'expliquer les ensembles de périodes d'exécution d'un groupe d'*Atomes* pour lesquels la stabilité du système est garantie. Ces ensembles de périodes peuvent donc devenir une entrée pour les algorithmes de l'*autonomic computing* afin de leur permettre de savoir quelle est la marge de manœuvre pour adapter les périodes. La décomposition en *Atomes* permet en outre aux experts du domaine de pouvoir opérer les adaptations nécessaires avec un grain plus fin que ce qui est possible avec une conception monolithique (i.e. pouvoir, par exemple, sur certains cycles d'exécution désactiver certains *Atomes* pour alléger la charge calculatoire instantanée).

Il est également nécessaire d'envisager de compléter notre approche de *Contraintes* supplémentaires qui pourraient permettre de mieux faire interagir les concepteurs de l'architecture de contrôle avec les spécialistes de l'*autonomic computing*.

Comme nous l'avons expliqué précédemment, notre formalisme s'incrit dans un cadre com-

plémentaire d'autres travaux orientés connaissances tels que les ontologies ou les DSLs qui représentent les connaissances à un niveau d'abstraction plus élevé, chaque approche pouvant de fait capturer des informations que peut difficilement représenter l'autre. En effet, dans le cadre de nos travaux, les ontologies et DSLs permettraient de modéliser des connaissances implicites et des hypothèses de conception qui sont difficiles à exprimer avec le formalisme *Atome*.

En effet, les conditions d'utilisation ou de validité des *Atomes* ne sont jamais explicitement représentées. Elles peuvent parfois être exprimées de manière indirecte via l'*Entité Composable* de sélection d'une *Alternative* mais ne sont jamais explicitement liées à l'une des entités que contient celle-ci. De plus, d'autres conditions sont très difficiles à représenter sous une forme exploitable calculatoirement. Considérons par exemple l'*Atome* $VirtualProximeter$ présenté au Chapitre 10. Cet *Atome* compense l'absence de mesures entre deux échantillons acquis à faible fréquence en supposant que l'environnement est constant entre ces deux mesures. D'autres *Atomes* jouant ce rôle pourraient encapsuler une connaissance plus complexe, permettant une estimation plus fine. L'hypothèse de "constance de l'environnement" est donc très forte mais n'est pas formellement descriptible dans notre approche (elle peut être au mieux décrite dans la documentation de l'*Atome*) alors qu'elle pourrait être capturée via une ontologie.

Un autre aspect de la complémentarité avec les ontologies est le fait de pouvoir décrire des conditions de validité sur des assemblages d'*Atomes* et de pouvoir vérifier leur compatibilité dans la *Composition*. Ainsi, certaines conditions d'utilisation ne proviennent pas d'une *Entité Composable* spécifique mais de la manière dont elles sont assemblées. Par exemple, la conception de notre algorithme de centrage impose que le robot soit capable de se déplacer suivant les axes $\vec{y_B}$ et $\vec{z_B}$ et que ses déplacements soient découplés entre ces axes et les autres degrés de liberté du robot. Pouvoir formaliser cette hypothèse permettrait donc à l'utilisateur de vérifier de manière systématique si un robot est compatible ou non avec une telle approche.

Dès lors, il est intéressant de travailler sur l'articulation entre notre approche et celles basées sur les ontologies et les DSLs afin de profiter de leur complémentarité pour aboutir à des travaux ayant une meilleure expressivité pour leurs utilisateurs.

Une autre perspective importante est celle de l'amélioration de la valuation des *Contraintes* temporelles notamment des durées d'exécution. Dans nos travaux, nous avons considéré le pire temps d'exécution de chaque *Atome*. Il s'agit naturellement d'une approche très conservative car la probabilité que chaque entité s'exécute à son pire cas temporel lors du même cycle d'exécution est assez faible comme souligné dans [GMG+15]. Cette approche trop conservative

peut être néfaste à la conception du robot car elle peut pousser le concepteur à surdimensionner certains composants (notamment les calculateurs) ce qui a un impact (négatif) en termes de coût ou de consommation énergétique. En outre, notre estimation des temps de calcul est limitée car, dans la manière assez simple dont nous l'avons effectué durant ces travaux, elle néglige de nombreux aspects de l'implémentation (dont les latences internes à l'OS temps-réel) qui peuvent grandement impacter les durées d'exécution. Il faudrait donc pouvoir raffiner ce processus.

Pour cela, il est nécessaire d'articuler nos travaux avec d'autres approches orientées vers la problématique complexe d'estimation du pire temps d'exécution. Ces approches sont généralement basées sur des données expérimentales recueillies lors de l'exécution du système [WR09], [ZBK11] ou [DP05] mais ces approches basées sur la stimulation du système par des vecteurs de test peinent à avoir une couverture exhaustive prenant en compte toutes les interactions dans un logiciel temps-réel. Ainsi, certaines approches probabilistes appliquées aux résultats temporels d'exécution ouvrent des perspectives intéressantes dans l'estimation du pire cas d'exécution, notamment celles basées sur la théorie de la valeur extrême [EB01], [HHM09] ou [MNS13] ou encore celles qui étendent la notion de pire temps d'exécution [GDLS15].

Si notre formalisation actuelle des *Contraintes* doit permettre une articulation relativement efficace avec ces approches, quelques modifications sémantiques ou dans la représentation des *Contraintes* pourraient s'avérer nécessaires.

Un autre sujet d'intérêt concerne le cas des architectures locales distribuées. En effet, dans de nombreux cas les calculs peuvent être répartis entre différents calculateurs suivant les besoins applicatifs. Dans notre cas, nous avons par exemple réparti les calculs de notre algorithme de centrage entre le PC de supervision (en guise de carte de haut-niveau) et la BeagleBone qui ne pouvait répondre seule à nos besoins. Dans les expérimentations présentées au Chapitre 10 nous avons négligé l'impact de la délocalisation des calculs (notamment au niveau des délais induits), car le faible volume de données échangées et la fréquence faible des échanges le permettaient. Néanmoins, nous voulons pouvoir estimer, dans un cas général, l'impact de la distribution des calculs et réifier son impact sur les *Contraintes* temporelles (notamment via les délais de communication). Une façon de représenter ces délais sans impacter le formalisme actuel pourrait être d'introduire des *Atomes* "délai" dont la *propriété* $t_{comp_{max}}$ représenterait le délai induit. Néanmoins, cette solution a le défaut majeur d'aller à l'encontre de notre volonté de proposer une description de nos *Compositions* qui soit indépendante de la cible d'implémentation. En effet, le positionnement des *Atomes* "délai" serait naturellement fonction de

la répartition des opérations entre les différents calculateurs ce qui est fondamentalement dépendant de l'architecture de calcul (nombre de processeurs, puissance de chacun). Dès lors, le formalisme et la méthodologie proposés devraient soit expliciter l'étape de déploiement et ainsi permettre d'introduire automatiquement ces *Atomes* "délai" selon la distribution choisie des calculs sur les différents nœuds, soit expliciter ces délais via le raffinement des *Contraintes* temporelles. La seconde option conduirait à une décomposition du terme t_{comm} présent dans les équations de diffusion des *Contraintes* (6.1) et (6.2). En effet, ce terme est à l'heure actuelle trop agrégé pour pouvoir précisément expliciter tous les phénomènes apparaissant au niveau des *Liens* entre *Atomes*.

Une extension de cette problématique concerne la coopération multi robots. En effet, outre les délais d'échange d'informations entre robots, les communications (surtout en milieu sous-marin) sont incertaines (perte d'échantillons voire perte de liaison). Il faut donc pouvoir exprimer son incertitude et évaluer l'impact de celle-ci sur les performances et la stabilité du contrôle afin de guider les concepteurs vers le choix des lois de commande les mieux adaptées pour faire face aux incertitudes de communication. Il s'agit là d'une réflexion théorique à mener à plus long terme, pouvant s'appuyer sur les nombreux travaux relatifs aux systèmes distribués.

Les perspectives précédentes ont mis en lumière la nécessité d'enrichir les *Contraintes* existantes. Nous avons largement évoqué la recherche d'une expression plus approfondie des *Contraintes* temporelles car celles-ci ont à la fois un impact important sur une propriété essentielle des lois de commande, leur stabilité, et un impact clé sur la structure du logiciel de contrôle. Toutefois, il faut aussi envisager l'intégration de *Contraintes* portant sur d'autres aspects afin de répondre aux besoins des divers utilisateurs. Il s'agit d'une réflexion sur le long terme à mener entre les différents acteurs du processus de développement d'un robot afin d'extraire des propriétés essentielles pour eux.

Ensuite, il est naturellement important de réfléchir à la manière d'intégrer ces nouvelles *Contraintes* à notre méthodologie et notamment de savoir si la phase d'étude présentée au Chapitre 6 peut être généralisée aux différentes *Contraintes* ou si chaque type de *Contrainte* requiert, de par ses spécificités, une phase d'étude indépendante.

Ainsi, notre formalisme n'est qu'une première étape et l'effort entrepris doit être poursuivi afin d'aboutir à une formalisation plus riche, précise et complète et doit naturellement s'accompagner du développement de la méthodologie associée à ce formalisme.

Annexe A

Illustration des concepts liés aux Atomes

Dans cette annexe, nous allons illustrer de manière détaillée et sur des exemples concrets les différents concepts liés aux *Atomes* que nous avons définis tout au long du Chapitre 4.

Reprenons les différentes connaissances mises en évidence dans la Figure 4.5 présentée Section 4.1.2. Nous allons montrer comment celles-ci sont traduites sous forme d'*Atomes*.

Exemple A.1:

Considérons l'*Atome Force to 0* (Figure 4.5, Blocs G et H et Equations (4.5) et (4.6)) repris dans la Figure A.1. Il est défini comme le 6-uplet :

$$Ft0 = (Nom(Ft0), Int(Ft0), Phy(Ft0), IntPar(Ft0), Ctrs(Ft0), KD(Ft0)) \tag{A.1}$$

conformément à la Définition 17.

Dans la suite, les différents *Eléments d'Interface* seront notés sous la forme :

$$Type < Frame, Axis > nom$$

Nous avons :

$$Nom(Ft0) = "ForceTo0" \tag{A.2}$$

$$KD(Ft0) = Task \tag{A.3}$$

En effet, cet *Atome* décrivant une équation de commande, il appartient au *Domaine de Connaissance Task*.

Les *Paramètres d'Interface* sont définis, conformément à Définition 15, par :

$$IntPar(Ft0) = \{Ent \in Fr, ax \in Ax\} \tag{A.4}$$

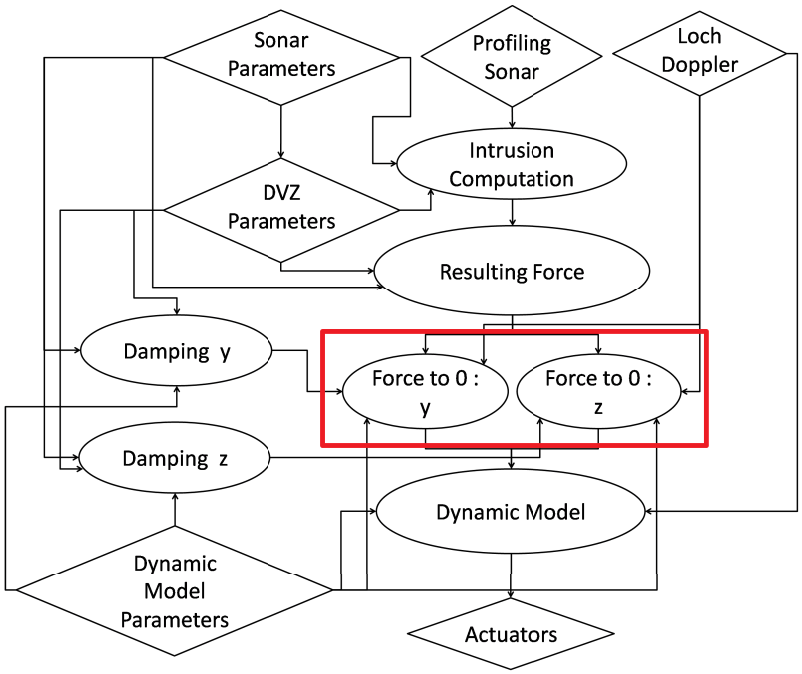

FIGURE A.1 – Connaissances traitées dans l'exemple A.1 (encadrées en rouge)

Ent sert à indiquer dans quel repère est effectué le contrôle puisque la connaissance décrite dans cet *Atome* ne s'attache pas à un repère particulier. *ax* sert à préciser l'axe dans lequel il est effectué.

Nous pouvons ainsi définir $Int(Ft0) = (Ne(Ft0), Pr(Ft0), IS(Ft0))$ comme :

$$IS(Ft0) = \emptyset \tag{A.5}$$

$$Ne(Ft0) = Mass < Ent, ax > m \tag{A.6}$$

$$Speed < Ent, ax > spe \tag{A.7}$$

$$Force < Ent, ax > F \tag{A.8}$$

$$Damping < Ent, ax > c \tag{A.9}$$

$$Pr(Ft0) = Acceleration < Ent, ax > accel \tag{A.10}$$

Cet exemple illustre bien le rôle joué par les *Paramètres d'Interface* qui assurent la cohérence d'utilisation des différents *Eléments d'Interface*. Dans notre cas, ils permettent de s'assurer que ceux-ci appartiendront tous au même repère et au même axe, permettant de se prémunir contre une mauvaise utilisation de l'*Atome* (i.e. des mauvaises liaisons avec d'autres *Atomes*).

La *Physique* est, conformément à Définition 16, l'application mathématique :

$$Phy(Ft0) : Mass < Ent, ax > \times Speed < Ent, ax > \times Force < Ent, ax > \times Damping < Ent, ax > \rightarrow Acceleration < Ent, ax > \quad (A.11)$$

définie par :

$$accel = \frac{F}{m} - \frac{c * spe}{m} \quad (A.12)$$

Enfin, tous les *Besoins* sont *Obligatoires*.

Exemple A.2:

Considérons l'*Atome* correspondant à la connaissance *Resulting Force* (Figure 4.5, Bloc F et Equations (4.3) et (4.4)) et repris à la Figure A.2, il est défini comme le 6-uplet :

$$RFC = (Nom(RFC), Int(RFC), Phy(RFC), IntPar(RFC), Ctrs(RFC), KD(RFC)) \quad (A.13)$$

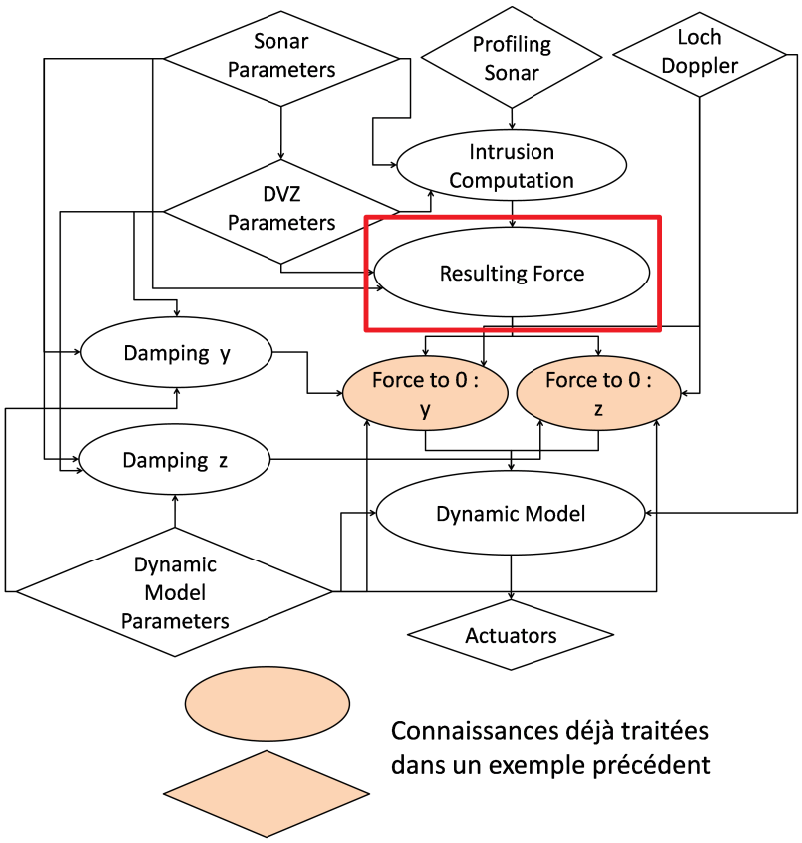

FIGURE A.2 – Connaissances traitées dans l'exemple A.2 (encadrées en rouge)

Nous avons :

$$Nom(RFC) = "ResultingForceComputation" \tag{A.14}$$

$$KD(RFC) = Environment :: Karst \tag{A.15}$$

Cet *Atome* décrit un phénomène d'interaction entre le robot et son environnement. Comme ce modèle a été conçu originellement pour les environnements karstiques, nous avons choisi de le rattacher au *Domaine de Connaissance* $Environment :: Karst$.

Les *Paramètres d'Interface* sont définis par :

$$IntPar(RFC) = \{Ent \in Fr, ax1 \in Ax, ax2 \in Ax\} \tag{A.16}$$

On peut ainsi définir $Int(RFC) = (Ne(RFC), Pr(RFC), IS(RFC))$ comme :

$$\begin{aligned}
IS(RFC) &= \emptyset && \text{(A.17)}\\
Ne(RFC) &= UInteger < NOFRAME, NOAXIS > Npts && \text{(A.18)}\\
&\quad Stiffness < NOFRAME, NOAXIS > KDVZ && \text{(A.19)}\\
&\quad Array < CylindricPoint < Ent >> intru && \text{(A.20)}\\
Pr(RFC) &= Force < Ent, ax1 > F1 && \text{(A.21)}\\
&\quad Force < Ent, ax2 > F2 && \text{(A.22)}
\end{aligned}$$

Dans ce cas $ax1$ et $ax2$ indiquent les axes selon lesquels seront exprimées les forces à annuler. L'élément *intru* est un *Type* array (puisque les intrusions sont exprimées sous forme d'un tableau de taille $Npts$). Le *Type* complexe *CylindricPoint* sert à représenter un point exprimé en coordonnées cylindriques. Plus de détails sur les *Types* actuellement définis et leur signification sont disponibles à l'Annexe A.

Ici un point exprimé en coordonnées cylindriques n'étant pas rattaché à un axe spécifique, il n'est défini que par un *repère*. Ce sera le cas de tous les *Types* complexes servant à définir des systèmes de coordonnées (qui sont pour l'instant les seules sortes de *Types* complexes définis).

Sa *Physique* est l'application mathématique :

$$\begin{aligned}
Phy(RFC) : &UInteger < NOFRAME, NOAXIS >\\
&\times Stiffness < NOFRAME, NOAXIS > \times Array < CylindricPoint < Ent >>\\
&\rightarrow Force < Ent, ax1 > \times Force < Ent, ax2 >
\end{aligned} \tag{A.23}$$

définie par :

$$F1 = 0$$
$$F2 = 0$$
$$for(i = 0; i < Npts; i++)$$
$$F1 = F1 - KDVZ * cos(intru[i].theta) * intru[i].r$$
$$F2 = F2 - KDVZ * sin(intru[i].theta) * intru[i].r$$

Enfin, tous les *Besoins* sont *Obligatoires*.

Exemple A.3:

Considérons l'*Atome* correspondant à la connaissance *Intrusion Computation* (Figure 4.5, Bloc E) repris dans la Figure A.3, il est défini comme le 6-uplet :

$$CDVZ = (Nom(CDVZ), Int(CDVZ), Phy(CDVZ), IntPar(CDVZ), Ctrs(CDVZ), KD(CDVZ)) \quad \text{(A.24)}$$

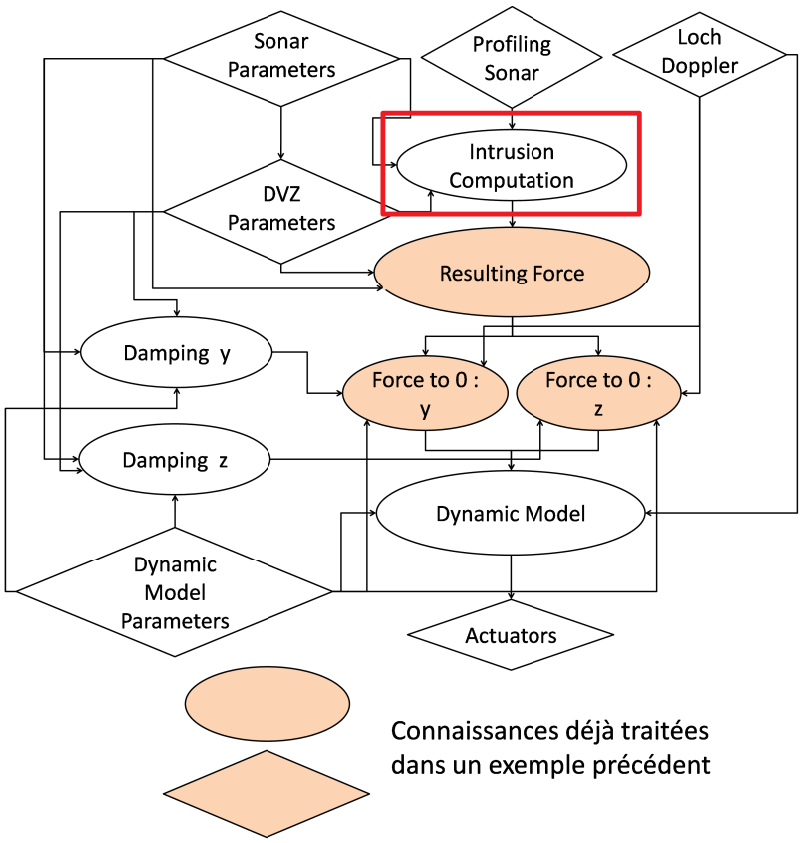

FIGURE A.3 – Connaissances traitées dans l'exemple A.3 (encadrées en rouge)

Nous avons :

$$Nom(CDVZ) = "CircularDVZ" \tag{A.25}$$

$$KD(CDVZ) = Environment :: Karst \tag{A.26}$$

Dans ce cas nous avons rattaché le calcul d'intrusion à la forme de la Zone Déformable Virtuelle (*Deformable Virtual Zone*, DVZ) qui est ici circulaire. En effet, il est possible de choisir des formes différentes pour celle-ci, ce qui va influencer la manière de calculer les intrusions et donc la valeur de la force résultante.

Les *Paramètres d'Interface* sont définis par :

$$IntPar(CDVZ) = \emptyset \tag{A.27}$$

On peut ainsi définir $Int(CDVZ) = (Ne(CDVZ), Pr(CDVZ), IS(CDVZ))$ comme :

$$\begin{aligned} IS(CDVZ) &= \emptyset && \text{(A.28)} \\ Ne(CDVZ) &= UInteger < NOFRAME, NOAXIS > Npts && \text{(A.29)} \\ & \quad Distance < DVZ, xy > RDVZ && \text{(A.30)} \\ & \quad Array < CylindricPoint < DVZ >> meas && \text{(A.31)} \\ Pr(CDVZ) &= Array < CylindricPoint < DVZ >> intru && \text{(A.32)} \end{aligned}$$

Ceci est le cas d'un *Atome* n'ayant pas de *Paramètres d'Interface*. A l'exception de la taille des vecteurs d'impacts et d'intrusion ($Npts$), toutes les données sont exprimées dans un repère dédié à la DVZ. En effet, si dans la Figure 4.4 la DVZ est représentée comme étant concomitante au plan $\{yz\}$ du robot et centrée sur celui-ci, il ne s'agit que d'un cas particulier car rien ne nous empêche de déplacer l'origine de cette zone par rapport au centre du robot. Cela pourrait même être souhaitable pour pouvoir déplacer le robot par rapport au point "sûr" du conduit bien que cette fonctionnalité doive être utilisée avec de grandes précautions dans un environnement karstique.

Dans la Section 4.1.2, nous avons considéré que la loi de commande centrait le robot. Or cette propriété n'est valable que dans le cas particulier où le centre de la zone virtuelle est identique au centre du robot. Mais en réalité c'est bien cette zone qui est centrée et comme elle est positionnée d'une manière fixe par rapport au robot, c'est ce dernier qui doit être déplacé pour assurer le déplacement de la DVZ. Nous voyons bien que ce fonctionnement n'apparait pas explicitement dans la version monolithique de la loi de commande présentée dans les Equations (4.1) et (4.2) mais se révèle bien plus clair avec l'utilisation des *Atomes* et la spécification des repères lors de leur composition comme nous le verrons dans l'Exemple B.1.

Sa *Physique* est l'application mathématique :

$$Phy(DVZ) : UInteger < NOFRAME, NOAXIS > \times Distance < DVZ, xy > \\ \times Array < CylindricPoint < DVZ >> \rightarrow Array < CylindricPoint < DVZ >> \quad (A.33)$$

définie par :

$$\begin{aligned} &for(i = 0; i < Npts; i++) \\ &\quad intru[i].r = RDVZ - meas[i].r \\ &\quad intru[i].theta = meas[i].theta \\ &\quad intru[i].h = meas[i].h \\ &\quad intru[i].or\_y = meas[i].or\_y \\ &\quad intru[i].or\_z = meas[i].or\_z \end{aligned}$$

Enfin, tous les *Besoins* sont *Obligatoires*.

Exemple A.4:

Considérons l'*Atome* correspondant à la connaissance *Dynamic Model* (Figure 4.5, Bloc L) repris dans la Figure A.4, il est défini comme le 6-uplet :

$$DMS = (Nom(DMS), Int(DMS), Phy(DMS), IntPar(DMS), Ctrs(DMS), KD(DMS)) \quad (A.34)$$

L'*Atome* que nous présenterons ici n'est qu'une version simplifiée du modèle dynamique du robot qui sera présenté de manière plus détaillée dans la Chapitre 9. Nous avons :

$$Nom(DMS) = "DynamicModelSimplified" \quad (A.35)$$

$$KD(DMS) = Technology :: Robot \quad (A.36)$$

Le modèle dynamique d'un robot représente l'interaction entre celui-ci et son environnement. Nous pouvons donc dire qu'il est à cheval entre les *Domaines de Connaissance Environment* et $Technology :: Robot$. Néanmoins, comme souligné dans la Section 4.2.5, l'expertise sur ce modèle est plutôt possédée par les roboticiens et il est donc plus logique de le rattacher au *Domaine de Connaissance* $Technology :: Robot$.

Les *Paramètres d'Interface* sont définis par :

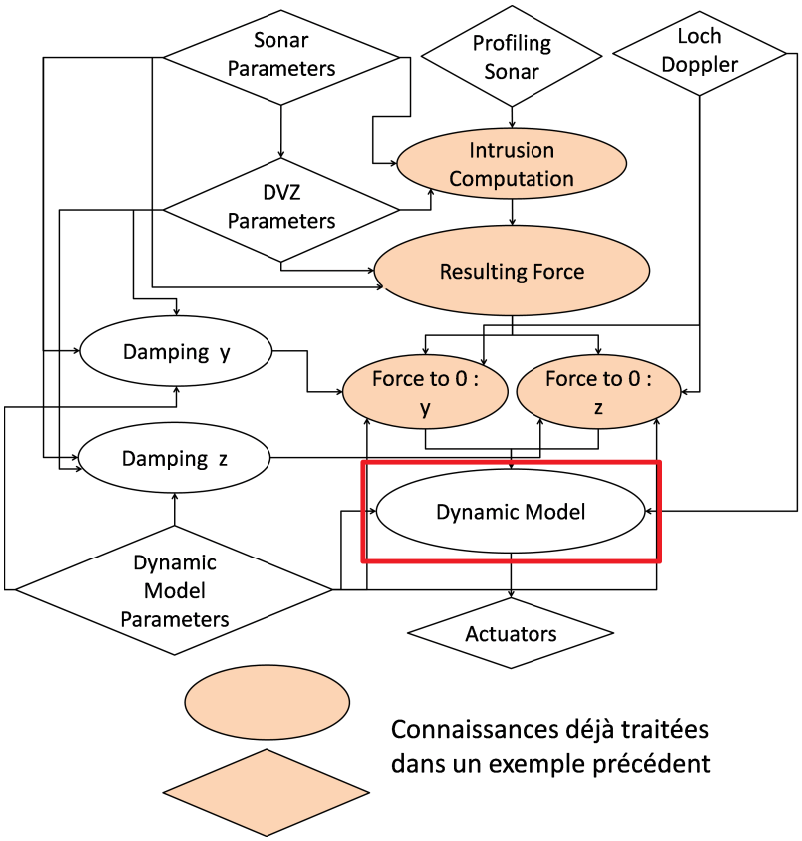

FIGURE A.4 – Connaissances traitées dans l'exemple A.4 (encadrées en rouge)

$$IntPar(DMS) = \{Ent \in Fr\} \tag{A.37}$$

On peut ainsi définir $Int(DMS) = (Ne(DMS), Pr(DMS), IS(DMS))$ comme :

$$\begin{aligned}
IS(DMS) &= \emptyset && \text{(A.38)}\\
Ne(DMS) &= Acceleration < Ent, y > dv && \text{(A.39)}\\
&\quad Acceleration < Ent, z > dw && \text{(A.40)}\\
&\quad Mass < Ent, y > mv && \text{(A.41)}\\
&\quad Mass < Ent, z > mw && \text{(A.42)}\\
&\quad Speed < Ent, y > v && \text{(A.43)}\\
&\quad Speed < Ent, z > w && \text{(A.44)}\\
&\quad Damping < Ent, y > d_v && \text{(A.45)}\\
&\quad Damping < Ent, z > d_w && \text{(A.46)}\\
Pr(DMS) &= Force < Ent, y > Fv && \text{(A.47)}\\
&\quad Force < Ent, z > Fw && \text{(A.48)}
\end{aligned}$$

Dans ce cas *Ent* sert à indiquer dans le repère de quelle entité (ici le robot) est représenté le modèle dynamique. Cela nous permet de souligner que, de manière générale, quand une connaissance est exprimée dans le repère robot, ce repère sera toujours représenté par un *Paramètre d'Interface* afin de permettre son utilisation simultanée par plusieurs robots si besoin est.

Sa *Physique* est l'application mathématique :

$$\begin{aligned} Phy(DMS) : Acceleration < Ent, y > \times Acceleration < Ent, z > \times Mass < Ent, y > \\ \times\, Mass < Ent, z > \times Speed < Ent, y > \times Speed < Ent, z > \times Damping < Ent, y > \\ \times\, Damping < Ent, z > \rightarrow Force < Ent, y > \times Force < Ent, z > \end{aligned} \quad (A.49)$$

définie par :

$$Fv = mv * dv - d\_v * v$$
$$Fw = mw * dw - d\_w * w$$

Enfin, tous les *Besoins* sont *Obligatoires*.

Exemple A.5:

Considérons maintenant les actionneurs de notre système (Figure 4.5, Bloc M) repris dans la Figure A.5. Leur pilotage est représenté par un *Atome* défini comme le 6-uplet :

$$ActS \quad = \quad (Nom(ActS), Int(ActS), Phy(ActS), IntPar(ActS), Ctrs(ActS), KD(ActS)) \quad (A.50)$$

Nous en présentons à nouveau, dans cet exemple, une version simplifiée par rapport à celle utilisée sur notre robot et qui est présentée Chapitre 9. On a :

$$Nom(ActS) = "ActuatorsSimplified" \quad (A.51)$$
$$KD(ActS) = Technology :: Actuators \quad (A.52)$$

S'agissant de l'*Atome* représentant l'actionnement du robot, il appartient naturellement au domaine $Technology :: Actuators$.

Les *Paramètres d'Interface* sont définis par :

$$IntPar(ActS) = \{Ent \in Fr\} \quad (A.53)$$

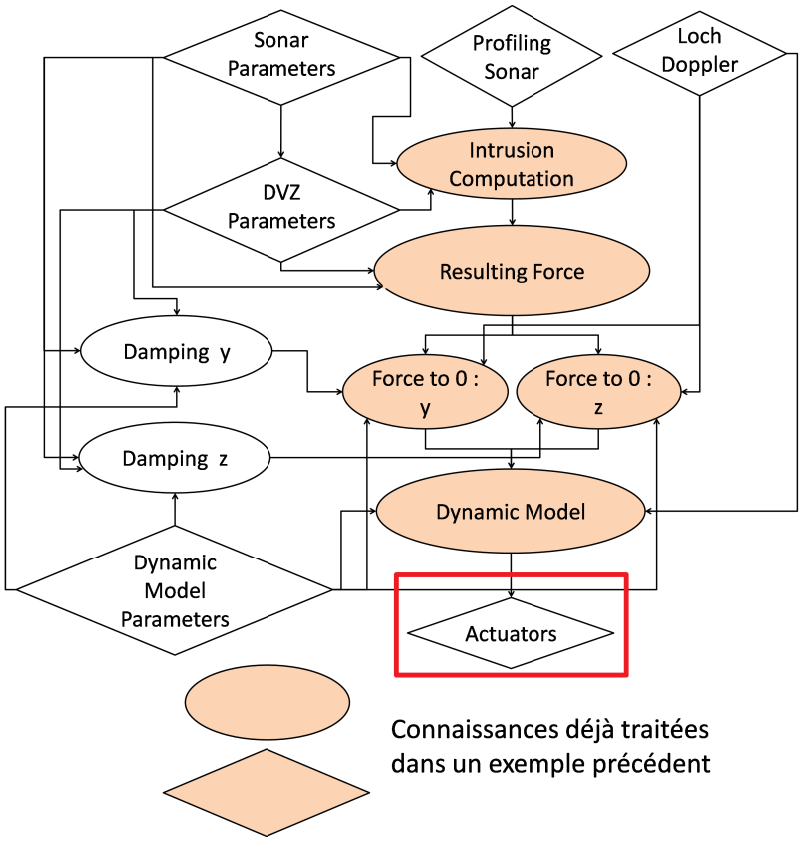

FIGURE A.5 – Connaissances traitées dans l'exemple A.5 (encadrées en rouge)

On peut ainsi définir $Int(ActS) = (Ne(ActS), Pr(ActS), IS(ActS))$ comme :

$$IS(ActS) = \emptyset \tag{A.54}$$

$$Ne(ActS) = Force < Ent, y > Fv \tag{A.55}$$

$$Force < Ent, z > Fw \tag{A.56}$$

$$Pr(ActS) = \emptyset \tag{A.57}$$

Ici, nous avons considéré un modèle très simple de l'*Atome* actionneur qui n'a ainsi que des *Besoins* qui matérialisent les forces que doivent appliquer les actionneurs. Il n'a donc pas de *Produits*.

De plus, cet *Atome* matérialise un point d'interaction entre notre architecture de contrôle et l'environnement du robot sur lequel vont agir les actionneurs du robot afin de déplacer ce dernier (voir Figure 4.10). Le contenu de sa *Physique* dépend donc de notre cible technologique (nombre de moteurs pour chaque degré de liberté, type de moteurs par exemple). Il s'agit donc d'une *Connaissance Externe* et, conformément à la Définition 25, sa *Physique* n'est pas définie :

$$Phy(ActS) = \emptyset \tag{A.58}$$

Enfin, tous les *Besoins* sont *Obligatoires*.

Exemple A.6:

Considérons maintenant le sonar profilométrique (Figure 4.5, Bloc A) repris dans la Figure A.6. Il est représenté par un *Atome* défini comme :

$$SonP = (Nom(SonP), Int(SonP), Phy(SonP), IntPar(SonP), Ctrs(SonP), KD(SonP)) \tag{A.59}$$

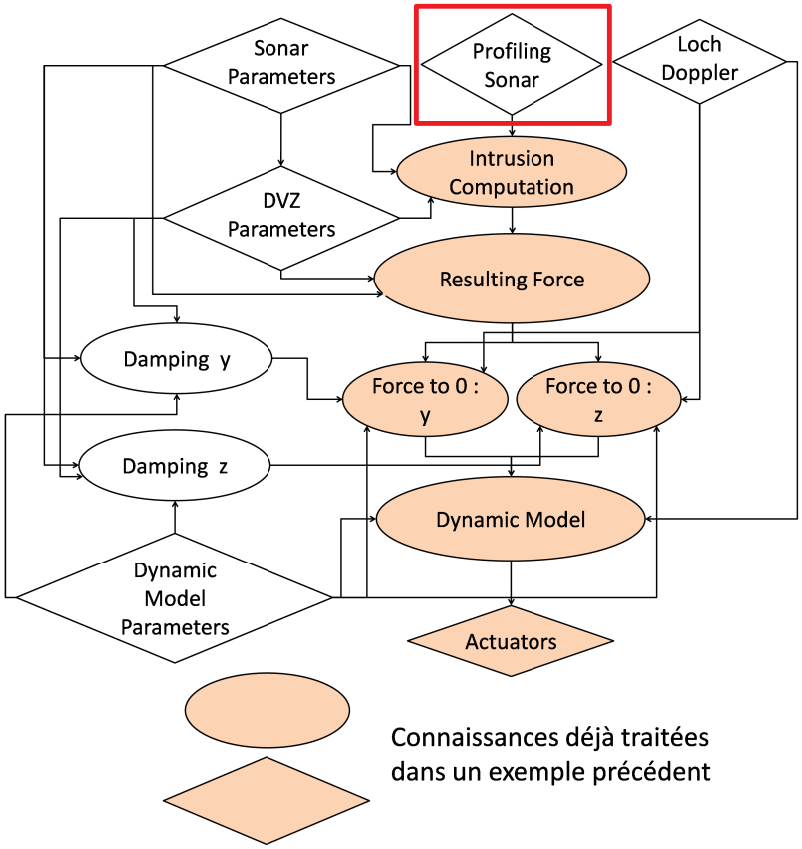

FIGURE A.6 – Connaissances traitées dans l'exemple A.6 (encadrées en rouge)

Nous avons :

$$Nom(SonP) = "ProfilingSonar" \tag{A.60}$$

$$KD(SonP) = Technology :: Sensors \tag{A.61}$$

S'agissant de l'*Atome* représentant un capteur, il appartient naturellement au domaine $Technology :: Sensors$.

Les *Paramètres d'Interface* sont définis par :

$$IntPar(SonP) = \emptyset \tag{A.62}$$

On peut ainsi définir $Int(SonP) = (Ne(SonP), Pr(SonP), IS(SonP))$ comme :

$$IS(SonP) = \emptyset \tag{A.63}$$
$$Ne(SonP) = \emptyset \tag{A.64}$$
$$Pr(SonP) = Array < CylindricPoint < PROXIMETER >> meas \tag{A.65}$$

Dans ce cas, nous avons considéré une version très simple du sonar qui se contente de renvoyer les impacts mesurés, dans le repère qui lui est propre. D'autres versions sont possibles pour, par exemple, réifier des points d'entrée pour le pilotage du sonar (changement de paramètres par exemple) ou encore pour indiquer, par exemple, le besoin d'un envoi régulier de données au sonar pour le maintenir actif. Ce cas là peut être intéressant pour réifier l'occurrence de ce phénomène régulier et le rendre explicite lors de l'étude des *Contraintes*.

Le sonar est aussi un point d'interaction entre notre architecture de contrôle et l'environnement dans lequel évolue le robot. Le contenu de sa *Physique* dépend donc de notre cible d'implémentation (concrètement le type ou la marque du sonar). Il s'agit donc d'une *Connaissance Externe* et, conformément à la Définition 25, sa *Physique* n'est pas définie :

$$Phy(SonP) = \emptyset \tag{A.66}$$

N'ayant pas de *Besoins*, cette entité n'a pas de *Contraintes* de connexion.

Exemple A.7:

Considérons maintenant le Loch Doppler (Figure 4.5, BlocB) repris dans la Figure A.7. Il est représenté par un *Atome* défini comme :

$$LDop = (Nom(LDop), Int(LDop), Phy(LDop), IntPar(LDop), Ctrs(LDop), KD(LDop)) \tag{A.67}$$

Nous avons :

$$Nom(LDop) = "LochDoppler" \tag{A.68}$$
$$KD(LDop) = Technology :: Sensors \tag{A.69}$$

S'agissant de l'*Atome* représentant un capteur, il appartient naturellement au domaine $Technology :: Sensors$.

Les *Paramètres d'Interface* sont définis par :

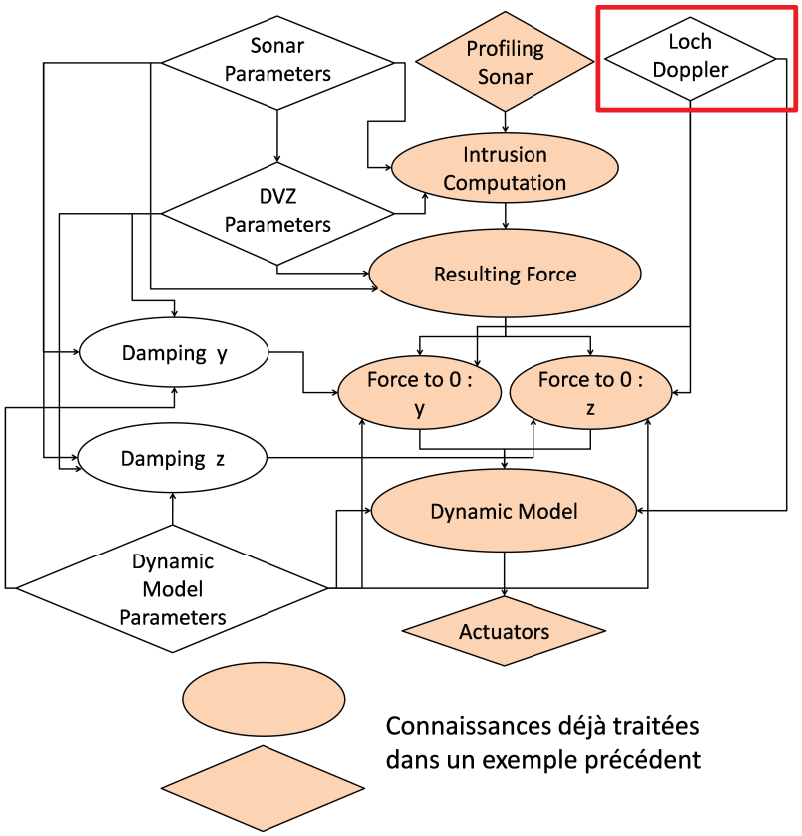

FIGURE A.7 – Connaissances traitées dans l'exemple A.7 (encadrées en rouge)

$$IntPar(LDop) = \emptyset \tag{A.70}$$

Nous pouvons ainsi définir $Int(LDop) = (Ne(LDop), Pr(LDop), IS(LDop))$ comme :

$$\begin{aligned} IS(LDop) &= \emptyset && \text{(A.71)} \\ Ne(LDop) &= \emptyset && \text{(A.72)} \\ Pr(LDop) &= Speed < LOCHDOPPLER, x > u && \text{(A.73)} \\ & \quad Speed < LOCHDOPPLER, y > v && \text{(A.74)} \\ & \quad Speed < LOCHDOPPLER, z > w && \text{(A.75)} \end{aligned}$$

Notre *Atome* fournit les mesures des vitesses suivant les trois axes. Celles-ci sont exprimées dans le repère du capteur, ici $LOCHDOPPLER$.

Tout comme le sonar, cet *Atome* réifie un point d'interaction entre notre architecture de contrôle et l'environnement dans lequel évolue le robot. Le contenu de sa *Physique* dépend donc de notre cible technologique (concrètement le type de capteur). Il s'agit donc d'une *Connaissance Externe* et, conformément à la Définition 25, sa *Physique* n'est pas définie :

$$Phy(LDop) = \emptyset \tag{A.76}$$

N'ayant pas de *Besoins*, cette entité n'a pas de *Contraintes* de connexion.

Nous n'avons pas présenté comment toutes les connaissances sont encapsulées dans des *Atomes*. Néanmoins, nous avons suffisamment d'*Atomes* pour illustrer la problématique de la liaison entre ceux-ci qui sera traitée dans l'Annexe B. De fait, afin de garder des explications simples en manipulant un nombre restreint d'*Entités Composables*, nous ne présenterons pas, pour l'instant, les *Atomes* contenant les connaissances non encore traitées.

Annexe B

Illustration des concepts liés à la Composition d'Entités Composables

Nous allons, dans cette annexe, illustrer les différents concepts liés aux *Molécules*, *Alternatives* et *Compositions* que nous avons introduits au Chapitre 5. Pour cela nous allons à nouveau nous inspirer de l'exemple proposé dans la Figure 4.5 (Chapitre 4) et notamment recourir aux entités déjà définies dans l'Annexe A.

Exemple B.1:

Nous n'avons pas encore présenté toutes les *Entités Composables* encapsulant les connaissances décrites dans la Figure 4.5. Néanmoins, nous pouvons, à partir des entités précédemment définies dans les Exemples A.1 à A.7, présenter un premier exemple de *Composition* qui va nous permettre d'illustrer les différents concepts liés à celles-ci.

Une *Composition* est définie comme un ensemble d'*Instances* et de *Liens* entre elles (Définition 31). Commençons donc par lister les *Instances* dont nous avons besoin.

Nous allons instancier le sonar profilométrique (Exemple A.6) comme $ProfilingSonar\ PS$. Il a pour *Contraintes* :

$$C_{t_t}(Phy(PS)) = Périodique$$

L'entité représentant le Loch Doppler (Exemple A.7) est instanciée comme $LochDoppler\ LD$. Elle a pour *Contraintes* :

$$C_{t_t}(Phy(LD)) = Périodique$$

L'*Instance* de l'*Atome* chargé du calcul des intrusions dans la zone virtuelle (Exemple A.3) est définie comme $CircularDVZ\ DVZ$ avec pour *Contraintes* :

$$C_{t_t}(Npts) = Sporadique$$
$$C_{t_t}(RDVZ) = Sporadique$$
$$C_{t_t}(meas) = Couplé$$
$$C_{t_t}(Phy(DVZ)) = Périodique$$

Nous avons ici considéré que les paramètres du sonar, ici $Npts$, pouvaient évoluer de manière *Sporadique* (par exemple changement du nombre de rayons du sonar par l'utilisateur) tout comme $RDVZ$ qui, s'il est un paramètre de la zone virtuelle, dépend d'un autre paramètre du sonar (Figure 4.5, Bloc D), sa portée d_{max} et prend par là même les contraintes émanant des paramètres sonar. A contrario, si nous avions estimé que ces paramètres ne pouvaient pas changer (i.e. paramétrage en ligne du sonar impossible) nous aurions alors choisi une *nature Constant* pour ces *Besoins*. Cet exemple nous permet ainsi d'insister sur le fait que les *natures* de *Contraintes temporelles* dépendent en grande partie du contexte d'utilisation de l'*Atome* et ne peuvent donc être définies que lors de son *Instanciation*, c'est-à-dire au moment où il va être utilisé dans une *Composition*. Les *Contraintes* permettent donc d'expliciter les choix de conception effectués et de vérifier leur compatibilité dans l'ensemble de la *Composition*. Cela nous permet également de mettre en évidence le fait que les entités composables peuvent être réutilisées dans des contextes applicatifs très différents.

L'*Instance* de l'*Atome* qui calcule la force résultante (Exemple A.2) est définie comme $ResultingForceComputation < Ent = DVZ, ax1 = y, ax2 = z > RFC$ avec pour *Contraintes* :

$$C_{t_t}(Npts) = Sporadique$$
$$C_{t_t}(KDVZ) = Constant$$
$$C_{t_t}(intru) = Couplé$$
$$C_{t_t}(Phy(RFC)) = Périodique$$

A l'instar de l'*Instance* précédente, nous avons donné une *nature Sporadique* aux *Besoins* qui sont en relation avec des paramètres du sonar mais, considérant que le paramétrage de la zone virtuelle ne pouvait pas évoluer, nous avons donné une *nature Constant* à ceux qui proviennent de la zone virtuelle, ici $KDVZ$.

Nous avons besoin de deux *Instances* pour annuler la force de réaction (Exemple A.1). La première est comme $ForceTo0 < Ent = ROBOT, ax = y > FT0y$ avec des *Contraintes* valuées comme :

$$C_{t_t}(m) = Constant$$
$$C_{t_t}(spe) = Couplé$$
$$C_{t_t}(F) = Couplé$$
$$C_{t_t}(c) = Sporadique$$
$$C_{t_t}(Phy(FT0y)) = Périodique$$

Nous avons ici supposé que les paramètres du robot, en l'occurrence sa masse dans l'eau, ne pouvaient pas changer en cours de mission. Par contre, le terme d'amortissement c peut lui changer de manière *Sporadique*. En effet, par la relation définie aux Equations (4.7) et (4.8) (Figure 4.5, Blocs I et J), c dépend du nombre de rayons de notre sonar, paramètre qui, comme nous l'avons spécifié précédemment, a été considéré comme susceptible de changer de valeur de manière *Sporadique*. De fait, la valeur de c est elle aussi susceptible d'être modifiée sporadiquement d'où le choix de *nature* de *Contraintes* effectué.

Similairement, nous définissons l'annulation de la composante suivant $\vec{z}_B$ de la force comme $ForceTo0 < Ent = ROBOT, ax = z > FT0z$ avec des *Contraintes* valuées comme :

$$C_{t_t}(m) = Constant$$
$$C_{t_t}(spe) = Couplé$$
$$C_{t_t}(F) = Couplé$$
$$C_{t_t}(c) = Sporadique$$
$$C_{t_t}(Phy(FT0z)) = Périodique$$

Il nous faut également instancier le modèle dynamique (Exemple A.4) comme $DynamicModelSimplified < Ent = ROBOT > DyMod$ avec des *Contraintes* valuées comme :

$$C_{t_t}(dv) = Périodique$$
$$C_{t_t}(dw) = Périodique$$
$$C_{t_t}(mv) = Constant$$
$$C_{t_t}(mw) = Constant$$
$$C_{t_t}(v) = Couplé$$
$$C_{t_t}(w) = Couplé$$
$$C_{t_t}(d\_v) = Constant$$
$$C_{t_t}(d\_w) = Constant$$
$$C_{t_t}(Phy(DyMod)) = Périodique$$

Ici, nous avons supposé que les paramètres du modèle dynamique (les masses totales du robot mv et mw ainsi que les amortissements d_v et d_w) ne pouvaient évoluer. Le choix des *Contraintes* implique en plus que les vitesses du robot, utilisées pour compenser le terme d'amortissement, doivent être mises à jour chaque fois que le modèle dynamique est calculé (d'où la *Contrainte* de *nature Couplé*). Par contre, les accélérations que le robot doit effectuer ne sont pas à mettre à jour à chaque calcul. En effet, le modèle dynamique d'un robot peut être interprété comme un asservissement en accélération dont les accélérations demandées, dv et dw, sont les consignes. De fait, il n'est pas nécessaire de mettre à jour celles-ci à chaque exécution de notre *Instance*. Comme les *Instances* déterminant la valeur de ces accélérations ($FT0y$ et $FT0z$) s'exécutent de manière périodique, leur valeur est mise à jour périodiquement et donc nous avons choisi des *Contraintes* temporelles de *nature Périodique*.

Enfin, l'entité pilotant les actionneurs (Exemple A.5) est instanciée comme $ActuatorsSimplified < Ent = ROBOT > Act$ avec des *Contraintes* valuées comme :

$$C_{t_t}(Fv) = Couplé$$
$$C_{t_t}(Fw) = Couplé$$
$$C_{t_t}(Phy(Act)) = Périodique$$

Ici, les consignes demandées aux actionneurs doivent être mises à jour avant chaque exécution de notre *Instance*.

Maintenant que nous avons défini les *Instances* utilisées, il faut lister les *Liens* entre elles.

Le premier d'entre eux connecte le *Produit meas* de PS, noté $PS.meas$, au *Besoin meas* de DVZ, noté $DVZ.meas$. Conformément à la Définition 26, ce lien est noté $L(PS.meas, DVZ.meas)$.

Conservant les mêmes conventions de notation, les autres *Liens* définis dans notre exemple sont : $L(DVZ.intru, RFC.intru)$, $L(RFC.F1, FT0y.F)$, $L(RFC.F2, FT0z.F)$, $L(FT0y.accel, DyMod.dv)$, $L(FT0z.accel, DyMod.dw)$, $L(DyMod.Fv, Act.Fv)$, $L(DyMod.Fw, Act.Fw)$, $L(LD.v, FT0y.spe)$, $L(LD.w, FT0z.spe)$, $L(LD.v, DyMod.v)$ et $L(LD.w, DyMod.w)$.

La *Composition* ainsi obtenue est représentée sous forme de *Graphe d'Association de Connaissances* (Définition 42) à la Figure B.1. Dans cette Figure, les *Liens* pointillés de couleur verte représentent les *Liens* vers un *Besoin* de *nature Périodique* et ceux en noir les *Liens* vers un *Besoin* de *nature Couplé*.

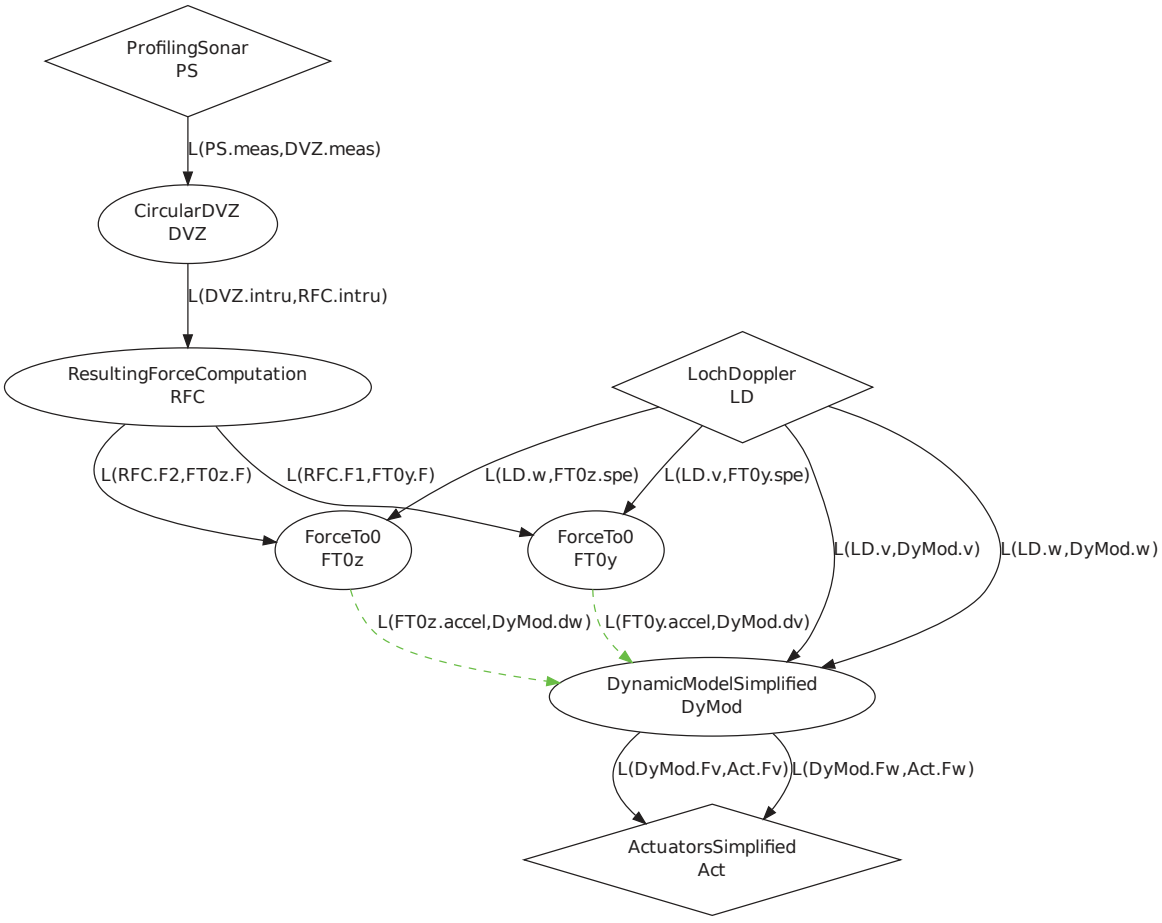

FIGURE B.1 – Représentation de la *Composition* exemple sous forme de *Graphe d'Association de Connaissances*

Il nous faut maintenant vérifier sa validité selon les conditions établies Définition 32. Aucune des *Instances* n'ayant de *Besoin* optionnel, la seconde condition est valide.

Considérons les *Besoins* de l'*Instance* DVZ. Nous avons :

$$fan\_in(Npts) = 0$$
$$fan\_in(RDVZ) = 0$$
$$fan\_in(meas) = 1$$

D'après la Définition 32, notre *Composition* n'est pas valide puisque certains *Besoins* Obligatoires ne sont pas connectés et ne satisfont donc pas aux *Contraintes* de connexion fixées. Néanmoins, afin d'éviter de saturer les explications de cet exemple en introduisant trop d'entités, nous considèrerons, dans un premier temps, que les connexions à effectuer d'après la Figure 4.5 le sont effectivement même si elles ne sont pas décrites dans notre *Composition*.

Nous devons désormais vérifier que nos *Liens* sont valides. Considérons le premier d'entre eux, $L(PS.meas, DVZ.meas)$. D'après la Définition 28, il est valide si et seulement si :

$$L(PS.meas, DVZ.meas) \in L_C\ (cohérence)$$
$$Ctrs(PS.meas) \equiv Ctrs(DVZ.meas)$$

Vérifions tout d'abord la compatibilité des *Contraintes*. Par la Définition 23, cela implique de vérifier la compatibilité des *Contraintes* temporelles. Ici, nous avons :

$$C_{t_t}(PS.meas) = C_{t_t}(Phy(PS)) = Périodique$$
$$C_{t_t}(DVZ.meas) = Couplé$$

D'après la Définition 22 et le Tableau 4.3, $C_{t_t}(PS.meas) \equiv C_{t_t}(DVZ.meas)$. Par la Définition 23, cela implique que les *Contraintes* sont compatibles.

Maintenant, afin d'assurer la cohérence du *Lien*, il faut vérifier que $D(PS.meas) = D(DVZ.meas)$ (Définition 27). De par la Définition 14, il nous faut donc nous assurer que :

$$D(PS.meas).Type = D(DVZ.meas).Type$$
$$D(PS.meas).Frame = D(DVZ.meas).Frame$$
$$D(PS.meas).Axis = D(DVZ.meas).Axis$$
$$size(D(PS.meas).Type) = size(D(DVZ.meas).Type)\ si D(PS.meas).Type \in Ty_{ar}$$

Nous avons $D(PS.meas).Type = Array < CylindricPoint >$ et $D(DVZ.meas).Type = Array < CylindricPoint >$ donc $D(PS.meas) = D(DVZ.meas)$. En outre, puisque nous avons affaire à des *Types* $Array$, $size(D(PS.meas).Type) = Npts$ et $size(D(DVZ.meas).Type) = Npts$ donc $size(D(PS.meas).Type) = size(D(DVZ.meas).Type)$.

Le *Type* complexe $CylindricPoint$ n'étant pas rattaché à un axe spécifique, la troisième condition est de fait valide.

Enfin, nous avons :

$$D(PS.meas).Frame = PROXIMETER$$
$$D(DVZ.meas).Frame = DVZ$$

Nous avons donc $D(PS.meas).Frame \neq D(DVZ.meas).Frame$. Le *Lien* n'est pas cohérent (Définition 14 et Définition 27). De par la Définition 32, cela signifie que notre *Composition* n'est pas valide. En effet, une donnée exprimée dans le repère de notre sonar $PROXIMETER$ et une exprimée dans le repère de notre zone virtuelle DVZ ne sont pas identiques. Il nous faut donc faire apparaître une ou plusieurs *Entités Composables* chargées d'effectuer le changement de repère. Celles-ci seront présentées dans les Exemples B.4 et B.5.

Avant de modifier notre *Composition*, vérifions la validité des autres *Liens* de manière identique à celle employée pour vérifier le *Lien* précédent. La Figure B.2 résume les liens valides et invalides.

Ainsi, les *Liens* entre le Loch Doppler et les *Atomes* utilisant les vitesses ne sont pas cohérents. En effet, si l'on se réfère aux différentes *Instances*, le Loch Doppler transmet les mesures dans son propre repère. De l'autre côté, les *Atomes* chargés de l'annulation de la Force et du modèle dynamique attendent des mesures dans le repère ROBOT. De fait, une connexion directe est impossible. Similairement, l'*Atome* chargé de calculer la force résultant des intrusions exprime les coordonnées de celle-ci dans le repère de notre zone virtuelle tandis que les *Atomes* chargés d'annuler cette force travaillent quant à eux dans le repère ROBOT. Encore une fois cela entraîne une incohérence du *Lien* qui n'est dès lors plus valide. Cet exemple illustre bien l'importance des données de repère et d'axe utilisées dans la description de l'*Interface* proposée dans le cadre de notre formalisme. En effet, elle nous a permis de détecter toutes les incohérences dans notre *Composition* nous permettant ainsi de les corriger.

Dans les Exemples B.2 à B.5, nous allons maintenant présenter les *Entités Composables* que nous allons ajouter pour permettre d'effectuer les changements de repère afin que notre *Composition* devienne valide. Nous reprendrons ensuite les modifications apportées à celle-ci à l'Exemple B.6.

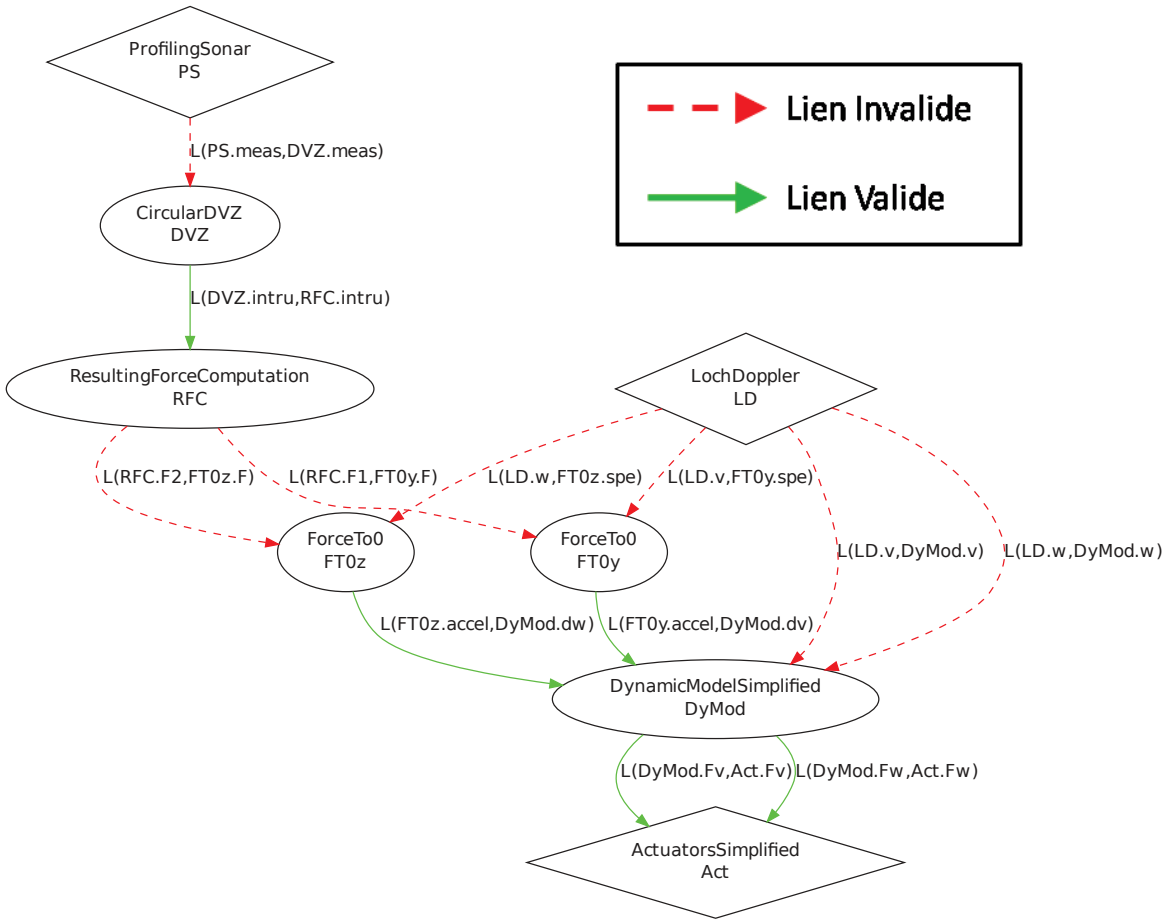

FIGURE B.2 – Liens valides et invalides dans notre *Composition* exemple

Exemple B.2:

L'*Entité Composable* chargée d'effectuer un changement de repère de Force est un *Atome* défini comme :

$$FSF = (Nom(FSF), Int(FSF), Phy(FSF), IntPar(FSF), Ctrs(FSF), KD(FSF)) \tag{B.1}$$

Nous avons :

$$Nom(FSF) = "FrameShiftForce" \tag{B.2}$$

$$KD(FSF) = Utilities :: MathUtils \tag{B.3}$$

Les *Paramètres d'Interface* sont définis par :

$$IntPar(FSF) = \{fr\_orig \in Fr, fr\_dest \in Fr\} \tag{B.4}$$

On peut ainsi définir $Int(FSF) = (Ne(FSF), Pr(FSF), IS(FSF))$ comme :

$$\begin{aligned} IS(FSF) &= \emptyset && \text{(B.5)} \\ Ne(FSF) &= Force < fr\_orig, x > Fx\_in && \text{(B.6)} \\ & \quad Force < fr\_orig, y > Fy\_in && \text{(B.7)} \\ & \quad Force < fr\_orig, z > Fz\_in && \text{(B.8)} \\ & \quad CartesianPose < fr\_dest > frame && \text{(B.9)} \\ Pr(FSF) &= Force < fr\_dest, x > Fx\_out && \text{(B.10)} \\ & \quad Force < fr\_dest, y > Fy\_out && \text{(B.11)} \\ & \quad Force < fr\_dest, z > Fz\_out && \text{(B.12)} \end{aligned}$$

Dans l'interface, $frame$ exprime les coordonnées du repère origine par rapport au repère destination. Il contient donc les trois coordonnées du centre du nouveau repère dans l'ancien ainsi que les rotations qu'ont subi les trois axes du repère.

Ici, les *Besoins* Fx_in, Fy_in et Fz_in sont Optionnels. Ainsi, nous ne sommes pas obligés de valuer les trois composantes de notre force pour effectuer le changement de repère. Dans les trois cas, leur valeur par défaut sera définie, via la *propriété* $default$ (Définition 20), comme $default = 0.0$.

Exemple B.3:

Le changement de repère des vitesses est réalisé par l'*Atome* défini comme :

$$FSS = (Nom(FSS), Int(FSS), Phy(FSS), IntPar(FSS), Ctrs(FSS), KD(FSS)) \quad \text{(B.13)}$$

On a :

$$\begin{aligned} Nom(FSS) &= "FrameShiftSpeed" && \text{(B.14)} \\ KD(FSS) &= Utilities :: MathUtils && \text{(B.15)} \end{aligned}$$

Les *Paramètres d'Interface* sont définis par :

$$IntPar(FSS) = \{fr\_orig \in Fr, fr\_dest \in Fr\} \quad \text{(B.16)}$$

On peut ainsi définir $Int(FSS) = (Ne(FSS), Pr(FSS), IS(FSS))$ comme :

$$IS(FSS) = \emptyset \quad \text{(B.17)}$$

$$Ne(FSS) = Speed < fr\_orig, x > u\_in \quad \text{(B.18)}$$

$$Speed < fr\_orig, y > v\_in \quad \text{(B.19)}$$

$$Speed < fr\_orig, z > w\_in \quad \text{(B.20)}$$

$$CartesianPose < fr\_dest > frame \quad \text{(B.21)}$$

$$Pr(FSS) = Speed < fr\_dest, x > u\_out \quad \text{(B.22)}$$

$$Speed < fr\_dest, y > v\_out \quad \text{(B.23)}$$

$$Speed < fr\_dest, z > w\_out \quad \text{(B.24)}$$

Tout comme pour l'*Atome* présenté à l'exemple précédent, $frame$ sert à décrire les coordonnées du repère origine par rapport au repère destination.

Les trois *Besoins* u_in, v_in et w_in sont Optionnels. Nous ne sommes donc pas obligés de valuer nos trois composantes de vitesse pour effectuer le changement de repère. Leur valeur par défaut est définie, via la *propriété* $default$ (Définition 20), comme $default = 0.0$.

Exemple B.4:

Considérons maintenant le changement de repère à effectuer entre le sonar et notre zone virtuelle. Dans notre cas, nous ne connaissons pas directement les coordonnées de l'un de ces repères par rapport à l'autre. Par contre, nous savons comment le sonar profilométrique est monté sur notre robot. Nous avons également choisi la position de notre zone virtuelle par rapport au robot. Nous allons donc découper notre phase de changement de repère en deux étapes. Nous transformerons dans un premier temps notre mesure dans l'espace sonar en mesure dans le repère robot. Nous effectuerons ensuite un changement de repère entre le repère robot et le repère DVZ.

Dans cet exemple, nous introduirons l'*Entité Composable* chargée d'effectuer le premier changement de repère puis présenterons l'autre entité dans l'exemple suivant.

Pour effectuer un changement de repère en coordonnées cylindriques, nous allons passer par trois étapes. La première consistera à exprimer nos points en coordonnées cartésiennes, nous effectuerons le changement de repère dans l'espace cartésien puis nous repasserons nos points en coordonnées cylindriques une fois la transformation effectuée. Cette entité est donc une *Molécule* contenant trois *Atomes*.

Le premier d'entre eux est *CylindricalToCartesian*. Il a pour *Paramètres d'Interface* et *Interface* :

$$IntPar(CylToCart) = \{Ent \in Fr\} \quad \text{(B.25)}$$

$$IS(CylToCart) = \emptyset \quad \text{(B.26)}$$
$$Ne(CylToCart) = UInteger < NOFRAME, NOAXIS > Npts \quad \text{(B.27)}$$
$$Array < CylindricPoint < Ent >> in \quad \text{(B.28)}$$
$$Pr(CylToCart) = Array < CartesianPose < Ent >> out \quad \text{(B.29)}$$

Une fois le changement de système de coordonnées effectué, nos points se retrouvent exprimés en coordonnées cartésiennes. $Npts$ fixe le nombre de points à traiter. Le *Paramètre d'Interface Ent* sert à indiquer le repère dans lequel s'effectue la transformation.

Le second *Atome* est $FrameShiftCartesian$ qui effectue le changement de repère en coordonnées cartésiennes. Il a pour *Paramètres d'Interface* et *Interface* :

$$IntPar(FSCart) = \{fr\_orig \in Fr, fr\_dest \in Fr\} \quad \text{(B.30)}$$

$$IS(FSCart) = \emptyset \quad \text{(B.31)}$$
$$Ne(FSCart) = UInteger < NOFRAME, NOAXIS > Npts \quad \text{(B.32)}$$
$$Array < CylindricPoint < fr\_orig >> CartIn \quad \text{(B.33)}$$
$$CartesianPose < fr\_dest > frame \quad \text{(B.34)}$$
$$Pr(FSCart) = Array < CartesianPose < fr\_dest >> CartOut \quad \text{(B.35)}$$

Similairement à l'Exemple B.2, $frame$ sert à indiquer les coordonnées du repère source dans le repère destination.

Enfin, *CartesianToCylindrical* est utilisé pour repasser les points en coordonnées cylindriques. Il a pour *Paramètres d'Interface* et *Interface* :

$$IntPar(CartToCyl) = \{Ent \in Fr\} \quad \text{(B.36)}$$

$$
\begin{aligned}
IS(CartToCyl) &= \emptyset && \text{(B.37)}\\
Ne(CartToCyl) &= UInteger < NOFRAME, NOAXIS > Npts && \text{(B.38)}\\
&\quad Array < CartesianPose < Ent >> in && \text{(B.39)}\\
&\quad Angle < Ent, y > or\_y && \text{(B.40)}\\
&\quad Angle < Ent, z > or\_z && \text{(B.41)}\\
Pr(CartToCyl) &= Array < CylindricPoint < Ent >> out && \text{(B.42)}
\end{aligned}
$$

Les *Besoins* or_y et or_z servent à contrôler l'orientation du cylindre par rapport au repère défini par *Ent* (voir Annexe C).

Nous pouvons maintenant définir notre *Molécule*. S'agissant d'une *Entité Composable*, elle est définie comme le 4-uplet (Définition 1) :

$$CyFS = (Nom(CyFS), Int(CyFS), Phy(CyFS), IntPar(CyFS)) \quad \text{(B.43)}$$

$$Nom(CyFS) = "CylindricalFrameShift" \quad \text{(B.44)}$$

Les *Paramètres d'Interface* sont définis par :

$$IntPar(CyFS) = \{fr\_orig \in Fr, fr\_dest \in Fr\} \quad \text{(B.45)}$$

Nous définissons ainsi $Int(CyFS) = (Ne(CyFS), Pr(CyFS), IS(CyFS))$ comme :

$$
\begin{aligned}
IS(CyFS) &= \emptyset && \text{(B.46)}\\
Ne(CyFS) &= UInteger < NOFRAME, NOAXIS > Npts && \text{(B.47)}\\
&\quad Array < CylindricPoint < fr\_orig >> Cylin && \text{(B.48)}\\
&\quad CartesianPose < fr\_dest > frame && \text{(B.49)}\\
&\quad Angle < fr\_dest, y > or\_y && \text{(B.50)}\\
&\quad Angle < fr\_dest, z > or\_z && \text{(B.51)}\\
Pr(CyFS) &= Array < CylindricPoint < fr\_dest >> Cylout && \text{(B.52)}
\end{aligned}
$$

Conformément à la Définition 33, la *Physique* est définie par le 4-uplet :

$$Phy(CyFS) = (E, L, l_n, l_p) \quad \text{(B.53)}$$

Lorsque nous définissons les entités contenues dans la *Physique* de notre *Molécule*, il nous faut leur donner un identifiant mais également lier leurs *Paramètres d'Interface* à ceux de la *Molécule* :

$$E = CylindricalToCartesian < Ent = fr\_orig > CylToCart \quad \text{(B.54)}$$
$$FrameShiftCartesian < fr\_orig = fr\_orig, fr\_dest = fr\_dest > FS \quad \text{(B.55)}$$
$$CartesianToCylindrical < Ent = fr\_dest > CartToCyl \quad \text{(B.56)}$$

L est l'ensemble des *Liens* défini comme :

$$L = L(CylToCart.out, FS.CartIn) \quad \text{(B.57)}$$
$$L(FS.CartOut, CartToCyl.in) \quad \text{(B.58)}$$

l_n représente l'ensemble des *Liens d'Interface* entre les *Besoins* de notre *Molécule* et ceux des entités qui sont contenues dans sa *Physique*, comme présenté Définition 34.

$$l_n = l(Npts, CylToCart.Npts) \quad \text{(B.59)}$$
$$l(Npts, FS.Npts) \quad \text{(B.60)}$$
$$l(Npts, CartToCyl.Npts) \quad \text{(B.61)}$$
$$l(or\_y, CartToCyl.or\_y) \quad \text{(B.62)}$$
$$l(or\_z, CartToCyl.or\_z) \quad \text{(B.63)}$$
$$l(Cylin, CylToCart.in) \quad \text{(B.64)}$$
$$l(frame, FS.frame) \quad \text{(B.65)}$$

l_p représente l'ensemble des *Liens d'Interface* entre les *Produits* de notre *Molécule* et ceux des entités qui sont contenues dans sa *Physique*.

$$l_p = l(Cylout, CartToCyl.out) \quad \text{(B.66)}$$

La Figure B.3 présente le *Graphe Moléculaire* associé à notre *Molécule*, tel que défini Définition 38.

Vérifions maintenant la validité de notre *Molécule* telle que posée à la Définition 36. Elle comprend les conditions suivantes :

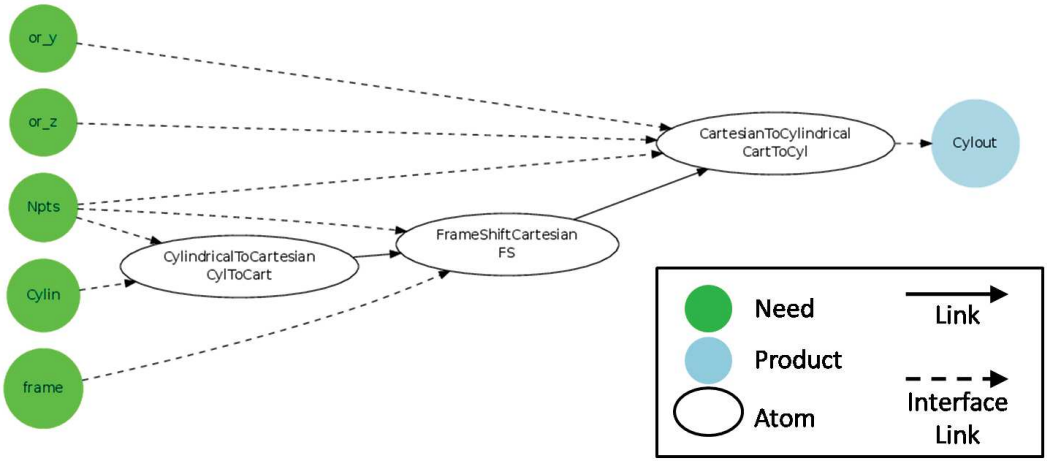

FIGURE B.3 – *Graphe Moléculaire* de la *Molécule CylindricalFrameShift*

$$fan\_in(i) = 1 \ \forall \ i \in Ob(E_j) \ \forall \ j \in [1 \ .. \ Dim(E)] \tag{B.67}$$

$$fan\_in(i) \leq 1 \ \forall \ i \in Op(E_j) \ \forall \ j \in [1 \ .. \ Dim(E)] \tag{B.68}$$

$$E_i \ est \ valide \ \forall i \in [1 \ .. \ Dim(E)] \tag{B.69}$$

$$L_i \in L_C \ \forall \ i \in [1 \ .. \ Dim(L)] \tag{B.70}$$

$$l_{n_i} \in l_C \ \forall \ i \in [1 \ .. \ Dim(l_n)] \tag{B.71}$$

$$l_{p_i} \in l_C \ \forall \ i \in [1 \ .. \ Dim(l_p)] \tag{B.72}$$

$$Dim(Ne(m)) \leq \sum_{i=1}^{Dim(E)} Dim(Ne(E_i)) \tag{B.73}$$

$$Dim(Pr(m)) \leq \sum_{i=1}^{Dim(E)} Dim(Pr(E_i)) \tag{B.74}$$

Les trois entités contenues dans cette *Molécule* sont toutes des *Atomes* et n'ont que des *Besoins* Obligatoires. Les conditions (B.68) et (B.69) sont donc vérifiées. On a $\sum_{i=1}^{Dim(E)} Dim(Ne(E_i)) = 9$ et $Dim(Ne(m)) = 5$ donc la condition (B.73) est vérifiée. De même, $\sum_{i=1}^{Dim(E)} Dim(Pr(E_i)) = 3$ et $Dim(Pr(m)) = 1$ donc (B.74) est vérifiée. En outre, tous les *Besoins* des différentes entités sont connectés une et une seule fois, (B.67) est donc vérifiée.

Assurons-nous de la cohérence des *Liens* en se basant sur la Définition 27. Nous avons (en fonction des *Paramètres d'Interface* de notre *Molécule*) :

$$D(CylToCart.out).Type = Array < CartesianPose >$$
$$D(FS.CartIn).Type = Array < CartesianPose >$$
$$D(FS.CartOut).Type = Array < CartesianPose >$$
$$D(CartToCyl.in).Type = Array < CartesianPose >$$
$$D(CylToCart.out).Frame = fr\_orig$$
$$D(FS.CartIn).Frame = fr\_orig$$
$$D(FS.CartOut).Frame = fr\_dest$$
$$D(CartToCyl.in).Frame = fr\_dest$$

Les différents *Types* sont complexes et n'ont donc pas de champ *Axis* associé. En outre, il s'agit de *Types Array* qui ont tous une taille identique, $Npts$. $L(CylToCart.out, FS.CartIn)$ et $L(FS.CartOut, CartToCyl.in)$ sont donc cohérents et la condition (B.70) est vérifiée.

Vérifions la cohérence des *Liens d'Interface* qui, comme présenté Définition 35, est identique à la vérification de cohérence des autres *Liens*. Considérons par exemple $l(Npts, CylToCart.Npts)$. Nous avons :

$$D(Npts).Type = UInteger$$
$$D(CylToCart.Npts).Type = UInteger$$
$$D(Npts).Frame = NOFRAME$$
$$D(CylToCart.Npts).Frame = NOFRAME$$
$$D(Npts).Axis = NOAXIS$$
$$D(CylToCart.Npts).Axis = NOAXIS$$

De fait, $D(Npts) = D(CylToCart.Npts)$ et $l(Npts, CylToCart.Npts)$ est donc cohérent. Nous pouvons vérifier similairement la cohérence des autres *Liens d'Interface* et donc vérifier la validité des conditions (B.71) et (B.72). Ainsi, d'après la Définition 36, notre *Molécule* est valide.

Exemple B.5:

Considérons maintenant le passage du repère robot vers le repère zone virtuelle. Contrairement au cas précédent, nous connaissons les coordonnées de la zone virtuelle par rapport au repère robot. Il s'agit donc d'une transformation inverse à la précédente.

Similairement à l'exemple précédent, nous allons convertir les coordonnées cylindriques en coordonnées cartésiennes puis effectuer le changement de repère avant de finalement repasser en coordonnées cylindriques. Nous retrouvons donc les deux *Atomes* définis à l'exemple

précédent, $CylindricalToCartesian$ et $CartesianToCylindrical$. Nous définissons un dernier *Atome* $FrameShiftInvCartesian$ qui a pour *Interface* et *Paramètres d'Interface* :

$$IntPar(FSICart) = \{fr\_orig \in Fr, fr\_dest \in Fr\} \tag{B.75}$$

$$\begin{aligned} IS(FSICart) &= \emptyset && \text{(B.76)}\\ Ne(FSICart) &= UInteger < NOFRAME, NOAXIS > Npts && \text{(B.77)}\\ & \quad Array < CylindricPoint < fr\_orig >> CartIn && \text{(B.78)}\\ & \quad CartesianPose < fr\_orig > frame && \text{(B.79)}\\ Pr(FSICart) &= Array < CartesianPose < fr\_dest >> CartOut && \text{(B.80)} \end{aligned}$$

La seule différence avec l'entité précédente est le *Paramètre d'Interface* du *Besoin* $frame$. Il sert à indiquer que nous exprimons les coordonnées du repère destination dans le repère origine.

Définissons maintenant notre *Molécule* comme le 4-uplet :

$$CyFSI = (Nom(CyFSI), Int(CyFSI), Phy(CyFSI), IntPar(CyFSI)) \tag{B.81}$$

$$Nom(CyFSI) = "CylindricalFrameShiftInv" \tag{B.82}$$

Les *Paramètres d'Interface* sont définis par :

$$IntPar(CyFSI) = \{fr\_orig \in Fr, fr\_dest \in Fr\} \tag{B.83}$$

Nous définissons ainsi $Int(CyFSI) = (Ne(CyFSI), Pr(CyFSI), IS(CyFSI))$ comme :

$$\begin{aligned} IS(CyFSI) &= \emptyset && \text{(B.84)}\\ Ne(CyFSI) &= UInteger < NOFRAME, NOAXIS > Npts && \text{(B.85)}\\ & \quad Array < CylindricPoint < fr\_orig >> Cylin && \text{(B.86)}\\ & \quad CartesianPose < fr\_orig > frame && \text{(B.87)}\\ & \quad Angle < fr\_dest, y > or\_y && \text{(B.88)}\\ & \quad Angle < fr\_dest, z > or\_z && \text{(B.89)}\\ Pr(CyFSI) &= Array < CylindricPoint < fr\_dest >> Cylout && \text{(B.90)} \end{aligned}$$

Conformément à la Définition 33, la *Physique* est définie par le 4-uplet :

$$Phy(CyFSI) = (E, L, l_n, l_p) \tag{B.91}$$

Les entités contenues dans notre *Molécule* sont :

$$E = CylindricalToCartesian < Ent = fr\_orig > CylToCart \tag{B.92}$$
$$FrameShiftInvCartesian < fr\_orig = fr\_orig, fr\_dest = fr\_dest > FS \tag{B.93}$$
$$CartesianToCylindrical < Ent = fr\_dest > CartToCyl \tag{B.94}$$

L est l'ensemble des *Liens* défini comme :

$$L = L(CylToCart.out, FS.CartIn) \tag{B.95}$$
$$L(FS.CartOut, CartToCyl.in) \tag{B.96}$$

l_n représente l'ensemble des *Liens d'Interface* entre les *Besoins* de notre *Molécule* et ceux des entités qui sont contenues dans sa *Physique*, comme présenté Définition 34.

$$l_n = l(Npts, CylToCart.Npts) \tag{B.97}$$
$$l(Npts, FS.Npts) \tag{B.98}$$
$$l(Npts, CartToCyl.Npts) \tag{B.99}$$
$$l(or\_y, CartToCyl.or\_y) \tag{B.100}$$
$$l(or\_z, CartToCyl.or\_z) \tag{B.101}$$
$$l(Cylin, CylToCart.in) \tag{B.102}$$
$$l(frame, FS.frame) \tag{B.103}$$

l_p représente l'ensemble des *Liens d'Interface* entre les *Produits* de notre *Molécule* et ceux des entités qui sont contenues dans sa *Physique*.

$$l_p = l(Cylout, CartToCyl.out) \tag{B.104}$$

La Figure B.4 présente le *Graphe Moléculaire* associé à notre *Molécule* tel que défini Définition 38.

Nous prouvons également sa validité similairement à l'exemple précédent.

Exemple B.6:

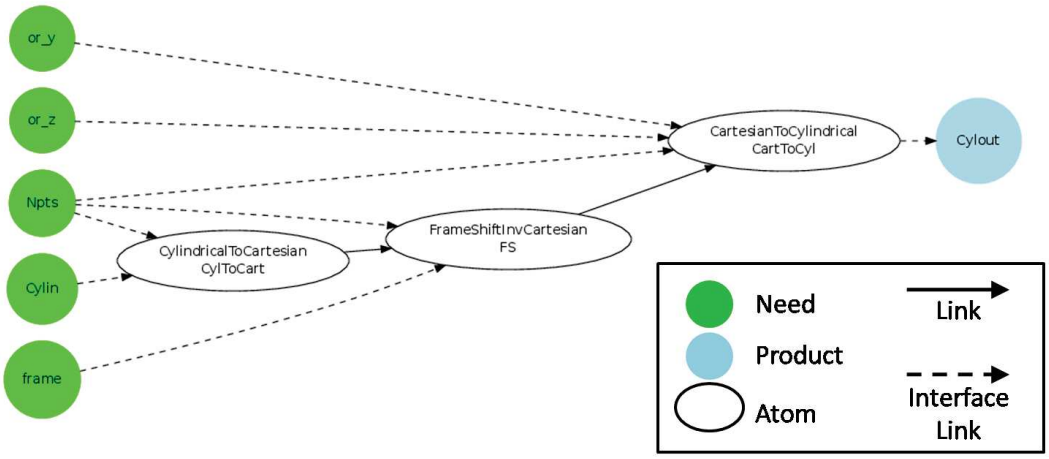

FIGURE B.4 – *Graphe Moléculaire* de la *Molécule* $CylindricalFrameShiftInv$

Nous avons maintenant ajouté les entités nous permettant de rendre la *Composition* présentée à l'Exemple B.1 valide. Instancions d'abord les entités définies aux Exemples B.2 à B.5.

L'*Instance* de l'*Atome* $FrameShiftForce$ (Exemple B.2) est définie comme :

$FrameShiftForce < fr\_orig = DVZ, fr\_dest = ROBOT > FToRob$ avec pour *Contraintes* :

$$C_{t_t}(Fy\_in) = Couplé$$
$$C_{t_t}(Fz\_in) = Couplé$$
$$C_{t_t}(frame) = Constant$$
$$C_{t_t}(Phy(FToRob)) = Périodique$$

Son *Besoin* Fx_in n'est pas utilisé, la *Contrainte* portant dessus n'a donc pas besoin d'être fixée. La *Contrainte* portant sur le *Besoin* $frame$ est $Constant$. Cela signifie que, dans notre exemple simple, nous supposons que la position de la zone virtuelle par rapport au robot ne peut pas être modifiée.

L'*Instance* de l'*Atome* $FrameShiftSpeed$ (Exemple B.3) est définie comme :

$FrameShiftSpeed < fr\_orig = LOCHDOPPLER, fr\_dest = ROBOT > LocToRob$

avec pour *Contraintes* :

$$C_{t_t}(u\_in) = Couplé$$
$$C_{t_t}(v\_in) = Couplé$$
$$C_{t_t}(w\_in) = Couplé$$
$$C_{t_t}(frame) = Constant$$
$$C_{t_t}(Phy(LocToRob)) = Périodique$$

La *Contrainte* portant sur le *Besoin* $frame$ est *Constant* puisque le montage du Loch Doppler sur le robot est fixe une fois pour toute.

Instancions maintenant $CylindricalFrameShift$. L'*Instance* est définie comme $CylindricalFrameShift < fr\_orig = PROXIMETER, fr\_dest = ROBOT > SonToRob$. Conformément à la Définition 37, les *Contraintes* portent sur les *Atomes* qu'elle contient. Celles-ci sont définies comme :

$$C_{t_t}(CylToCart.Npts) = Sporadique$$
$$C_{t_t}(CylToCart.in) = Couplé$$
$$C_{t_t}(Phy(CylToCart)) = Périodique$$

$$C_{t_t}(FS.Npts) = Sporadique$$
$$C_{t_t}(FS.CartIn) = Couplé$$
$$C_{t_t}(FS.frame) = Constant$$
$$C_{t_t}(Phy(FS)) = Périodique$$

$$C_{t_t}(CartToCyl.Npts) = Sporadique$$
$$C_{t_t}(CartToCyl.in) = Couplé$$
$$C_{t_t}(CartToCyl.or\_y) = Constant$$
$$C_{t_t}(CartToCyl.or\_z) = Constant$$
$$C_{t_t}(Phy(CartToCyl)) = Périodique$$

Instancions maintenant $CylindricalFrameShiftInv$. L'*Instance* est définie comme $CylindricalFrameShiftInv < fr\_orig = ROBOT, fr\_dest = DVZ > RobToDVZ$. Les *Contraintes* portant sur les *Atomes* qu'elle contient sont définies comme :

$$C_{t_t}(CylToCart.Npts) = Sporadique$$
$$C_{t_t}(CylToCart.in) = Couplé$$
$$C_{t_t}(Phy(CylToCart)) = Périodique$$

$$C_{t_t}(FS.Npts) = Sporadique$$
$$C_{t_t}(FS.CartIn) = Couplé$$
$$C_{t_t}(FS.frame) = Constant$$
$$C_{t_t}(Phy(FS)) = Périodique$$

$$C_{t_t}(CartToCyl.Npts) = Sporadique$$
$$C_{t_t}(CartToCyl.in) = Couplé$$
$$C_{t_t}(CartToCyl.or\_y) = Constant$$
$$C_{t_t}(CartToCyl.or\_z) = Constant$$
$$C_{t_t}(Phy(CartToCyl)) = Périodique$$

L'ensemble des *Liens* entre nos différentes *Instances* est défini comme :

$$
\begin{aligned}
L \;=\; & L(PS.meas, SonToRob.Cylin) && \text{(B.105)}\\
& L(SonToRob.Cylout, RobToDVZ.inCylin) && \text{(B.106)}\\
& L(RobToDVZ.Cylout, DVZ.meas) && \text{(B.107)}\\
& L(DVZ.intru, RFC.intru) && \text{(B.108)}\\
& L(RFC.F1, FToRob.Fy\_in) && \text{(B.109)}\\
& L(RFC.F2, FToRob.Fz\_in) && \text{(B.110)}\\
& L(FToRob.Fy\_out, FT0y.F) && \text{(B.111)}\\
& L(FToRob.Fz\_out, FT0z.F) && \text{(B.112)}\\
& L(FT0y.accel, DyMod.dv) && \text{(B.113)}\\
& L(FT0z.accel, DyMod.dw) && \text{(B.114)}\\
& L(DyMod.Fv, Act.Fv) && \text{(B.115)}\\
& L(DyMod.Fw, Act.Fw) && \text{(B.116)}\\
& L(LD.u, LocToRob.u\_in) && \text{(B.117)}\\
& L(LD.v, LocToRob.v\_in) && \text{(B.118)}\\
& L(LD.w, LocToRob.w\_in) && \text{(B.119)}\\
& L(LocToRob.v\_out, FT0y.spe) && \text{(B.120)}\\
& L(LocToRob.w\_out, FT0z.spe) && \text{(B.121)}\\
& L(LocToRob.v\_out, DyMod.v) && \text{(B.122)}\\
& L(LocToRob.w\_out, DyMod.w) && \text{(B.123)}
\end{aligned}
$$

La *Composition* ainsi obtenue est représentée sous forme de *Graphe d'Association de Connaissances* (Définition 42) à la Figure B.5.

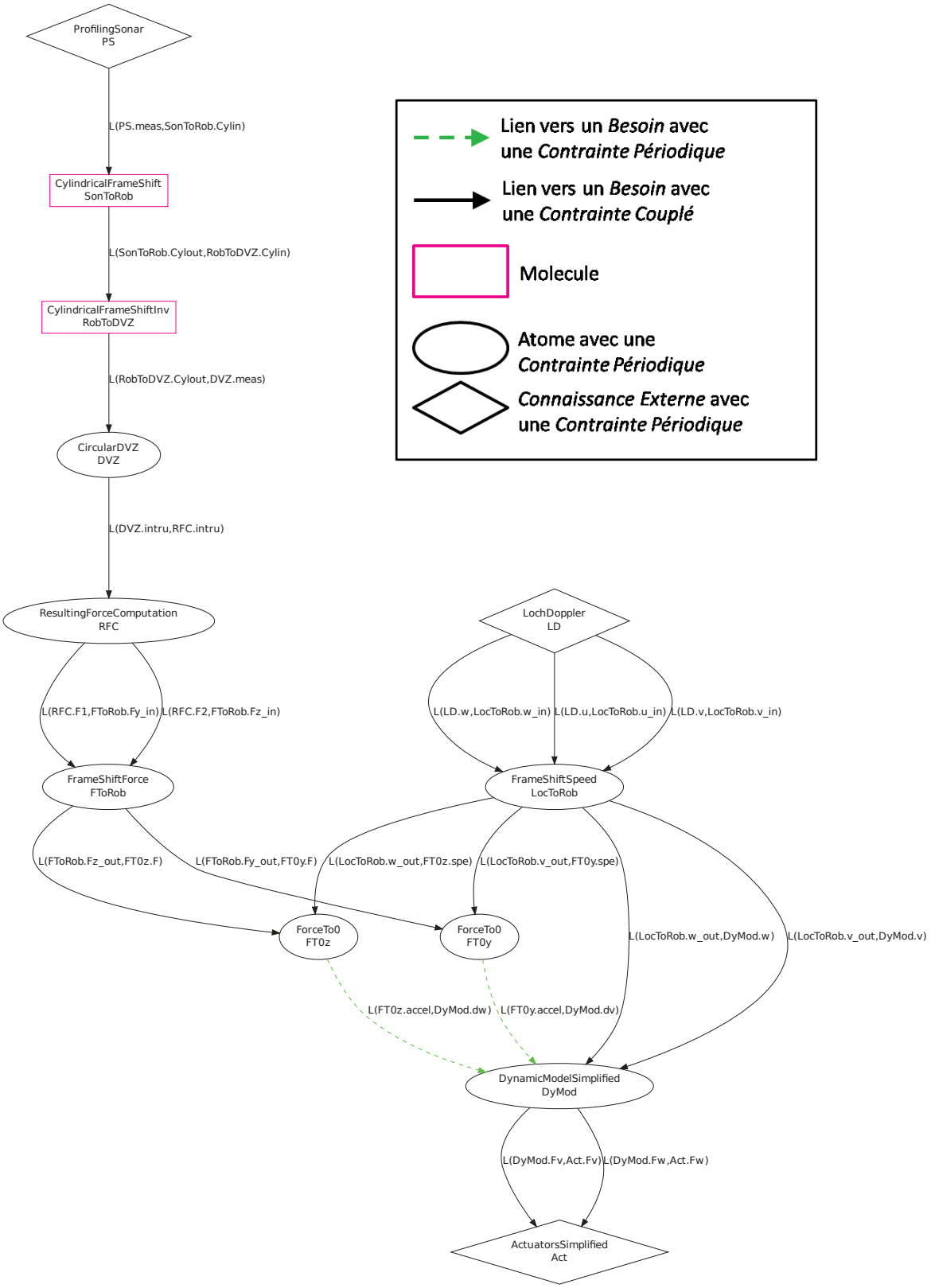

FIGURE B.5 – Représentation sous forme de *Graphe d'Association de Connaissances* de la *Composition* exemple modifiée pour être valide

Vérifions maintenant la validité du *Lien* $L(PS.meas, SonToRob.Cylin)$.

$$C_{t_t}(PS.meas) = C_{t_t}(Phy(PS)) = Périodique$$
$$C_{t_t}(SonToRob.Cylin) = C_{t_t}(SonToRob.CylToCart.in) = Couplé$$

Nous avons donc bien $C_{t_t}(PS.meas) \equiv C_{t_t}(SonToRob.Cylin)$. Il est ici intéressant de noter que les *Contraintes* ne portent pas directement sur l'*Instance* de notre *Molécule* mais, comme expliqué Définition 37, portent sur les *Atomes* que celle-ci contient. Pour vérifier la cohérence du *Lien*, nous sommes donc obligés de passer via le *Lien d'Interface* afin de trouver les *Besoins* auxquels sont liés le *Besoin* de notre *Molécule* (la *Contrainte* est donc "propagée" de l'*Atome* à la *Molécule* qui le contient). Cela permet aussi de constater que si plusieurs *Besoins* sont connectés au même *Besoin* d'une *Molécule*, cela impose que les *Contraintes* portant sur chacun d'entre eux soient identiques.

Vérifions maintenant la cohérence du *Lien*. On a donc, en se rappelant que l'*Instance* *SonToRob* a pour *Paramètres d'Interface* $< fr\_orig = PROXIMETER, fr\_dest = ROBOT >$:

$$D(PS.meas).Type = Array < CylindricPoint >$$
$$D(SonToRob.Cylin).Type = Array < CylindricPoint >$$
$$size(D(PS.meas).Type) = Npts$$
$$size(D(SonToRob.Cylin).Type) = Npts$$
$$D(PS.meas).Frame = PROXIMETER$$
$$D(SonToRob.Cylin).Frame = PROXIMETER$$

Nous avons donc bien

$$D(PS.meas).Type = D(SonToRob.Cylin).Type$$
$$size(D(PS.meas).Type) = size(D(SonToRob.Cylin).Type)$$
$$D(PS.meas).Frame = D(SonToRob.Cylin).Frame$$

De fait, par les Définitions 14, 23, 27 et 28, notre *Lien* est valide.

Similairement, nous pouvons montrer la validité des autres *Liens*. De fait, les liaisons de notre *Composition* sont devenues valides par l'ajout des *Entités Composables* chargées de réifier les changements de repère. Cela force le concepteur de la loi de commande à rendre

ceux-ci explicites, permettant ainsi une meilleure compréhension de la *Composition* et de ses intentions et choix de conception.

Cet exemple nous a également permis de montrer que la définition de l'*Interface*, dans notre formalisme, permet de détecter des incohérences et des erreurs au niveau des liaisons entre les différentes entités, ce qui n'est pas présent dans d'autres approches de la littérature telles que Modelica.

Exemple B.7:

Dans cet exemple, nous allons compléter la *Composition* précédente par les connaissances de la Figure 4.5 qui n'ont pas encore été traitées. Celles-ci sont rappelées à la Figure B.6.

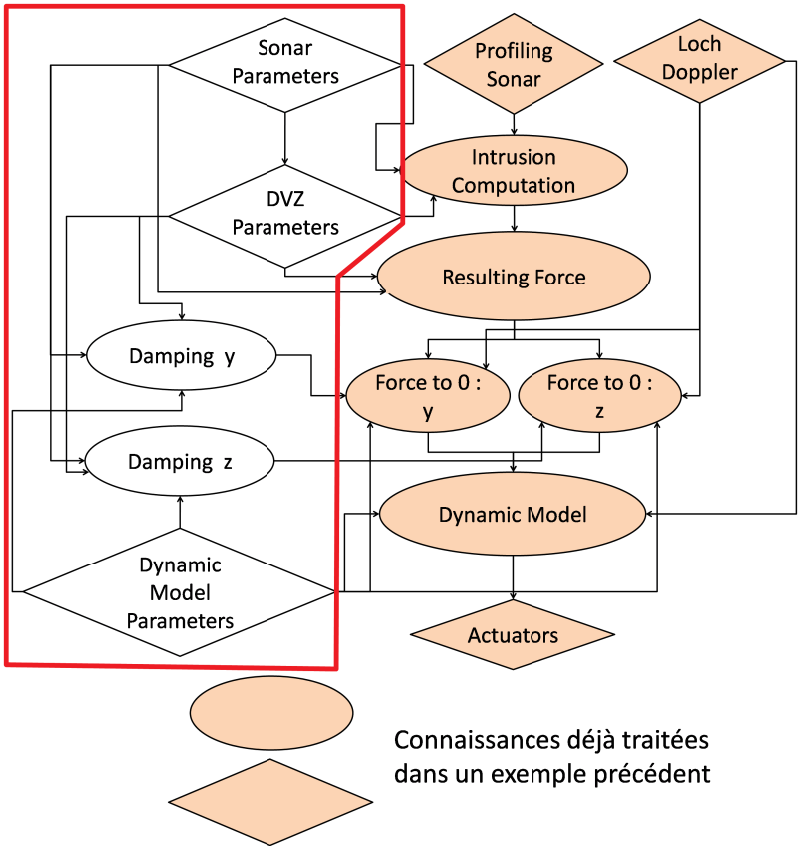

FIGURE B.6 – Connaissances traitées dans l'exemple B.7 (encadrées en rouge)

Commençons par la connaissance portant sur les paramètres du modèle dynamique (Figure 4.5, Bloc K). Etant donné que le modèle dynamique est simplifié, nous présenterons ici une forme simplifiée de cet *Atome*. Il s'agit d'$ExpertiseDynModSimplified$, une *Connaissance Externe*. Il a pour *Paramètres d'Interface* et *Interface* :

$$IntPar(ExpDMS) = \{Ent \in Fr\} \tag{B.124}$$

$$IS(ExpDMS) = \emptyset \quad \text{(B.125)}$$
$$Ne(ExpDMS) = \emptyset \quad \text{(B.126)}$$
$$Pr(ExpDMS) = Mass < Ent, y > mv \quad \text{(B.127)}$$
$$Mass < Ent, z > mw \quad \text{(B.128)}$$
$$Damping < Ent, y > d\_v \quad \text{(B.129)}$$
$$Damping < Ent, z > d\_w \quad \text{(B.130)}$$

Il s'agit d'un *Atome Expertise*. Il s'agit d'une famille d'*Atomes* qui ont un rôle précis et donc des caractéristiques et une implémentation similaires. Les *Atomes Expertise* ont pour rôle de représenter la fixation de la valeur de certains paramètres de notre commande par un "expert" (humain) à l'initialisation. De fait, il est possible de leur dégager certaines caractéristiques. Ils sont toujours des *Connaissances Externes*, n'ont jamais de *Besoin* (puisqu'il s'agit de "points d'entrée" pour l'opérateur humain) et sont toujours implémentés avec une *Contrainte* de *nature Constant*. Ainsi notre exemple est instancié comme :

$ExpertiseDynModSimplified < Ent = ROBOT > ExpDM$

avec pour *Contrainte* temporelle :

$$C_{t_t}(Phy(ExpDM)) = Constant$$

Considérons ensuite le calcul du coefficient d'amortissement afin de permettre une convergence en régime critique ((Figure 4.5, Blocs I et J). Dans les équations (4.7) et (4.8), la valeur de l'amortissement est calculée en fonction de la masse du robot, de la raideur affectée à notre zone virtuelle et du nombre de rayons du sonar. Or, le calcul de l'amortissement est un calcul d'ordre plus général qui ne dépend alors que de la masse et de la raideur du ressort. Dans un souci de modularité, nous allons donc diviser les calculs des équations (4.7) et (4.8) en deux parties, la première chargée de calculer l'amortissement pour un système masse-ressort-amortisseur général et la seconde chargée de calculer la raideur équivalente dans notre cas. Le calcul de l'amortissement est effectué par l'*Atome CriticalDamping* qui a pour *Paramètres d'Interface* et *Interface* :

$$IntPar(CrD) = \{Ent \in Fr, ax \in Ax\} \quad \text{(B.131)}$$

$$IS(CrD) = \emptyset \quad \text{(B.132)}$$
$$Ne(CrD) = Mass < Ent, ax > m \quad \text{(B.133)}$$
$$Stiffness < NOFRAME, NOAXIS > K \quad \text{(B.134)}$$
$$Pr(CrD) = Damping < Ent, ax > c \quad \text{(B.135)}$$

Sa *Physique* réalise les opérations suivantes :

$$c = 2 * sqrt(m * K)$$

La première instance est définie comme $CriticalDamping < Ent = ROBOT, ax = y > Damp\_y$, elle a pour *Contraintes* temporelles :

$$C_{t_t}(m) = Constant$$
$$C_{t_t}(K) = Sporadique$$
$$C_{t_t}(Phy(Damp\_y)) = Sporadique$$

Ici, m qui est la masse du robot est définie comme constante en adéquation avec l'entité précédemment définie. Comme l'exécution de la *Physique* est de *nature Sporadique*, il nous faut définir, comme présenté Tableau 4.2, l'entité chargée de décrire ses conditions d'exécution, ici un changement de valeur de K. Celles-ci sont définies par l'*Atome* $HasChangedStiffness$, qui a pour *Paramètres d'Interface* et *Interface* :

$$IntPar(HCS) = \{Ent \in Fr, ax \in Ax\} \quad \text{(B.136)}$$

$$IS(HCS) = Stiffness < Ent, ax > K\_prec \quad \text{(B.137)}$$
$$Ne(HCS) = Stiffness < Ent, ax > K \quad \text{(B.138)}$$
$$Pr(HCS) = Boolean < NOFRAME, NOAXIS > hasChanged \quad \text{(B.139)}$$

Ici, la valeur de K est comparée à celle, stockée, de K_prec et si elle est différente, la nouvelle valeur est stockée et $hasChanged$ passe à vrai, permettant d'exécuter la *Physique*.

La seconde instance $CriticalDamping < Ent = ROBOT, ax = z > Damp\_z$ porte des *Contraintes* identiques à la précédente.

Le calcul de la raideur équivalente est assuré par l'entité $DVZKeq$ qui n'a pas de *Paramètre d'Interface* et a pour *Interface* :

$$\begin{aligned} IS(CrD) &= \emptyset && \text{(B.140)} \\ Ne(CrD) &= UInteger < NOFRAME, NOAXIS > Npts && \text{(B.141)} \\ & \quad Stiffness < NOFRAME, NOAXIS > KDVZ && \text{(B.142)} \\ Pr(CrD) &= Stiffness < NOFRAME, NOAXIS > Keq && \text{(B.143)} \end{aligned}$$

Il est instancié comme $DVZKeq\ CompK$ avec pour *Contraintes* :

$$\begin{aligned} C_{t_t}(Npts) &= Sporadique \\ C_{t_t}(K) &= Constant \\ C_{t_t}(Phy(CompK)) &= Sporadique \end{aligned}$$

On a $C_{s_{phy}} = HasChangedUInteger < Ent = NOFRAME, ax = NOAXIS > C\_s\_phy$. Cet *Atome* est quasi identique à celui présenté précédemment, seul le *Datatype* sur lequel il porte change.

Les paramètres du sonar (Figure 4.5, Bloc B) sont rattachés à la *Connaissance Externe* $SonarParameters$. Elle n'a pas de *Paramètre d'Interface* et a pour *Interface* :

$$\begin{aligned} IS(SonP) &= \emptyset && \text{(B.144)} \\ Ne(SonP) &= \emptyset && \text{(B.145)} \\ Pr(SonP) &= UInteger < NOFRAME, NOAXIS > NR && \text{(B.146)} \\ & \quad UInteger < PROXIMETER, xy > range && \text{(B.147)} \end{aligned}$$

Elle est instanciée comme $SonarParameters\ ParSon$ avec pour *Contraintes* :

$$C_{t_t}(Phy(ParSon)) = Sporadique$$

Ici, il s'agit d'une entité avec *Physique* de *nature Sporadique* mais cela est fait pour décrire un phénomène extérieur non prévisible (ici, un changement de paramétrage de l'utilisateur) et dont les conditions d'exécution ne peuvent être données. Elle a ainsi pour $C_{s_{phy}}$, l'*Atome*

$TRUE$ qui a un seul *Produit* booléen qui vaut toujours vrai.

Les paramètres de la zone virtuelle sont fixés dans un *Atome Expertise*. Celui-ci est appelé $ExpertiseDVZ$, il a pour *Paramètres d'Interface* et *Interface* :

$$IntPar(ExpD) = \{Ent \in Fr\} \tag{B.148}$$

$$\begin{aligned}
IS(ExpD) &= \emptyset && \text{(B.149)}\\
Ne(ExpD) &= \emptyset && \text{(B.150)}\\
Pr(ExpD) &= Stiffness < NOFRAME, NOAXIS > KDVZ && \text{(B.151)}\\
&\quad CartesianPose < Ent > poseDVZ && \text{(B.152)}\\
&\quad Angle < DVZ, y > or_y && \text{(B.153)}\\
&\quad Angle < DVZ, z > or_z && \text{(B.154)}\\
& && \text{(B.155)}
\end{aligned}$$

Dans cette entité, $poseDVZ$ exprime le positionnement de la zone virtuelle dans le repère robot tandis que or_y et or_z servent à paramétrer l'orientation des coordonnées cylindriques dans le repère DVZ (voir Annexe C).

Il est instancié comme $ExpertiseDVZ < Ent = ROBOT > ExpDVZ$ avec pour *Contraintes* :

$$C_{t_t}(Phy(ExpDM)) = Constant$$

Enfin, nous ajoutons une *Molécule* chargée de regrouper les différents paramètres de la zone virtuelle, ceux fixés par l'*Expertise* précédente et ceux venant du paramétrage du sonar. Elle contient des *Atomes* identité et sert à découpler la valuation d'un paramètre de son utilisation afin d'apporter plus de modularité en cas de modification de la méthode de valuation de ces paramètres car cela évite de devoir refaire toutes les connexions entre eux et le reste de la *Composition*, simplifiant ainsi les modifications à apporter.

Il s'agit de la *Molécule* $DVZParameters$ qui est utilisée comme présenté Figure B.7.

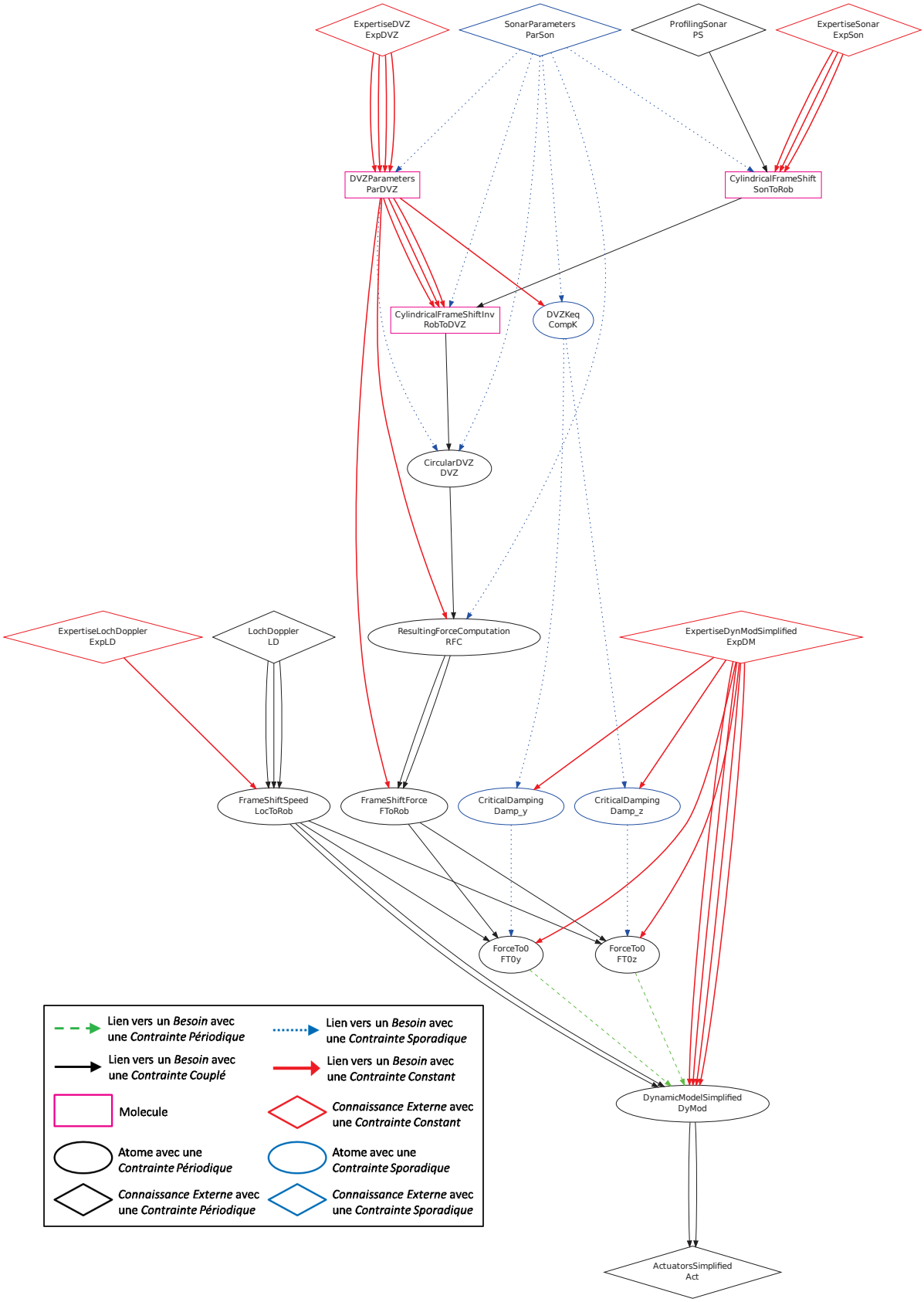

FIGURE B.7 – Représentation sous forme de *Graphe d'Association de Connaissances* de la *Composition* exemple enrichie des entités non périodiques

Enfin, nous ajoutons deux *Atomes Expertise* qui contiennent le montage du sonar profilométrique et du Loch Doppler sur le robot définissant ainsi les deux connaissances utilisées pour réaliser les changements de repère :

$$ExpertiseLochDoppler < Ent = ROBOT > ExpLD$$
$$ExpertiseSonar < Ent = ROBOT > ExpSon$$

Ils sont instanciés avec une *Physique* de *nature Constant* comme tous les *Atomes Expertise*.

La *Composition* complète est ainsi décrite Figure B.7.

Exemple B.8:

Illustrons maintenant les *Alternatives* via un exemple. Nous allons pour cela considérer un autre exemple applicatif que celui traité jusqu'à présent. Considérons le contrôle du cap d'un robot. Celui-ci peut soit être commandé directement par l'opérateur en téléopération, soit être asservi.

Définissons maintenant notre *Alternative* comme le 4-uplet :

$$AngM = (Nom(AngM), Int(AngM), Phy(AngM), IntPar(AngM)) \tag{B.156}$$

$$Nom(AngM) = "AngleManagement" \tag{B.157}$$

Les *Paramètres d'Interface* sont définis par :

$$IntPar(AngM) = \{Ent \in Fr, ax \in Ax\} \tag{B.158}$$

On peut ainsi définir $Int(AngM) = (Ne(AngM), Pr(AngM), IS(AngM))$ comme :

$$\begin{aligned}
IS(AngM) &= \emptyset && \text{(B.159)}\\
Ne(AngM) &= Angle < Ent, ax > angle_meas && \text{(B.160)}\\
&\quad Duration < NOFRAME, NOAXIS > t_meas && \text{(B.161)}\\
&\quad AngularSpeed < Ent, ax > spe_meas && \text{(B.162)}\\
&\quad Boolean < NOFRAME, NOAXIS > useControl && \text{(B.163)}\\
Pr(AngM) &= AngularAcceleration < Ent, ax > accel && \text{(B.164)}
\end{aligned}$$

Ici, nous avons décrit une *Alternative* permettant de gérer le contrôle ou la téléopération d'un angle. Ce seront à nouveau les *Paramètres d'Interface* qui nous permettront de déterminer l'angle concerné. Notre *Alternative* a quatre *Besoins*. Le premier, $angle\_meas$, représente la

mesure de l'angle, t_meas correspond à la date de la mesure (nécessaire pour les intégrations), spe_meas est la vitesse d'évolution de l'angle et $useControl$ est un signal booléen qui sera utilisé par l'entité de sélection de notre *Alternative* pour choisir entre la téléopération et l'asservissement. Le *Produit accel* est l'accélération angulaire à appliquer au robot.

Conformément à la Définition 40, la *Physique* est définie par le 7-uplet :

$$Phy(AngM) = (E, e_{sel}, E_{jun}, L_{sel}, L_{jun}, l_n, l_p) \tag{B.165}$$

Cette *Alternative* englobe deux entités. La première est une *Connaissance Externe* servant à représenter la téléopération du robot et nommée $TeleopAngularAccel$. Elle a pour *Paramètres d'Interface* et *Interface* :

$$IntPar(TelAngAcc) = \{Ent \in Fr, ax \in Ax\} \tag{B.166}$$

$$\begin{aligned}
IS(TelAngAcc) &= \emptyset && \text{(B.167)}\\
Ne(TelAngAcc) &= \emptyset && \text{(B.168)}\\
Pr(TelAngAcc) &= AngularAcceleration < Ent, ax > accel && \text{(B.169)}
\end{aligned}$$

Ici, les *Paramètres d'Interface* nous permettent de choisir dans quel repère et autour de quel axe sera appliquée l'accélération angulaire.

La seconde entité est une *Molécule*, $AngleControl$, qui a pour *Paramètres d'Interface* et *Interface* :

$$IntPar(AngCtrl) = \{Ent \in Fr, ax \in Ax\} \tag{B.170}$$

$$\begin{aligned}
IS(AngCtrl) &= Angle < Ent, ax > angle_ref && \text{(B.171)}\\
&\quad\ Angle < Ent, ax > error_tt && \text{(B.172)}\\
Ne(AngCtrl) &= Angle < Ent, ax > angle_meas && \text{(B.173)}\\
&\quad\ AngularSpeed < Ent, ax > spe_meas && \text{(B.174)}\\
&\quad\ Duration < NOFRAME, NOAXIS > t_meas && \text{(B.175)}\\
Pr(AngCtrl) &= AngularAcceleration < Ent, ax > accel && \text{(B.176)}
\end{aligned}$$

Elle est représentée par le *Graphe Moléculaire* défini Figure B.8.

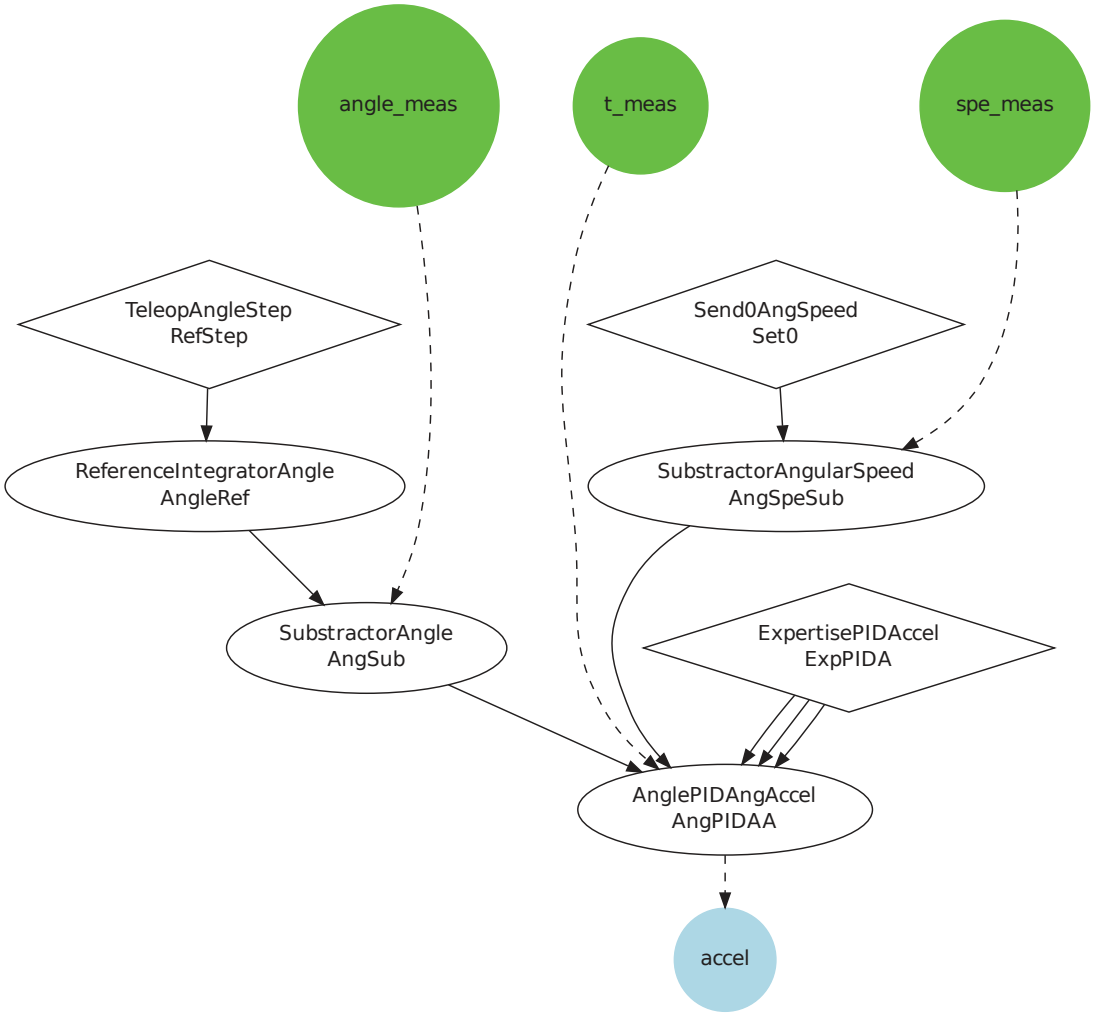

FIGURE B.8 – *Graphe Moléculaire* représentant la *Molécule AngleControl*

Il est à noter que l'ensemble des *Stockages Internes* d'une *Molécule* est l'union des *Stockages Internes* des entités qu'elle contient. Nous n'entrerons pas dans le détail de l'explication des différentes entités, ceci est fait au Chapitre 9.

Ainsi, on a :

$$E = TeleopAngularAccel < Ent = Ent, ax = ax > e1 \quad \text{(B.177)}$$

$$AngleControl < Ent = Ent, ax = ax > e2 \quad \text{(B.178)}$$

L'entité de sélection est l'*Atome SelectorAngleManagement e_sel*. Il n'a pas de *Paramètre d'Interface* et a pour *Interface* :

$$IS(e\_sel) = \emptyset \quad \text{(B.179)}$$

$$Ne(e\_sel) = Boolean < NOFRAME, NOAXIS > useControl \quad \text{(B.180)}$$

$$Pr(e\_sel) = TextString < NOFRAME, NOAXIS > entityToRun \quad \text{(B.181)}$$

Il s'agit d'un *Atome* sélecteur. C'est une famille d'*Atomes* qui sont utilisés pour décrire le processus de décision dans une *Alternative*. Ils ont toujours une unique sortie qui contient l'identifiant de l'entité à utiliser sous forme d'une chaîne de caractères. Ils ont également un certain nombre de *Besoins* tous booléens qui servent à exprimer les différentes conditions dictant le choix de l'entité à utiliser. Dans notre exemple simple, cet *Atome* est seul dans l'entité de sélection puisqu'il est directement connecté au signal booléen servant à choisir entre la téléopération ou l'asservissement. Mais dans un cas plus général, nécessitant de monitorer des connaissances comme nous le voyons au Chapitre 10, les *Atomes* sélecteurs seront accompagnés d'autres *Entités Composables* chargées d'effectuer ce monitoring.

Sa *Physique* est :

$$\begin{aligned} &if(useControl == TRUE) \\ &\quad entityToRun = "e2"; \\ &else \\ &\quad entityToRun = "e1"; \end{aligned}$$

Dans notre exemple, le processus décisionnel est limité à une instruction conditionnelle mais dans des situations plus complexes, il est possible de mettre en œuvre au sein des sélectionneurs des mécanismes décisionnels plus complexes pour, par exemple, gérer les conflits (i.e. plusieurs entités substituables dont les conditions d'exécution sont valides en même temps) tels que l'utilisation de tables de priorité.

Les entités de jonction sont définies comme :

$$E_{jun} = InitAngleControl < Ent = Ent, ax = ax > e12 \quad \text{(B.182)}$$

$$Empty\ e21 \quad \text{(B.183)}$$

L'entité $e21$ qui assure le passage de l'asservissement en cap à la téléopération est l'*Atome Vide*, nommé *Empty*. Cet *Atome* n'a ni *Besoins* ni *Produits* ni *Stockages Internes*. Il est néanmoins nécessaire d'utiliser cet *Atome* pour respecter la cardinalité de l'ensemble E_{jun} afin

de s'assurer la validité de l'*Alternative* permettant ainsi d'expliciter l'absence d'opérations lors de cette jonction. En outre, lors de l'étude des *Contraintes*, la présence de cet *Atome* permettra d'expliciter l'impact de la durée de commutation (liée aux mécanismes du *Middleware*) sur la faisabilité ou non du contrôle Figure B.9. Ainsi même s'il n'est pas connecté aux autres entités, sa présence est essentielle dans la vérification des propriétés temporelles du système.

Le passage de la téléopération à l'asservissement en cap est assuré par une *Molécule InitAngleControl*. Elle est chargée d'initialiser les *Stockages Internes* de la *Molécule AngleControl*. Elle a pour *Paramètres d'Interface* et *Interface* :

$$IntPar(InitAngCtrl) = \{Ent \in Fr, ax \in Ax\} \tag{B.184}$$

$$\begin{aligned}
IS(InitAngCtrl) &= \emptyset && \text{(B.185)}\\
Ne(InitAngCtrl) &= Angle < Ent, ax > angle_meas && \text{(B.186)}\\
&\quad TextString < NOFRAME, NOAXIS > entityToRun && \text{(B.187)}\\
Pr(InitAngCtrl) &= Angle < Ent, ax > angle_ref && \text{(B.188)}\\
&\quad Angle < Ent, ax > err_pid && \text{(B.189)}
\end{aligned}$$

Cette *Molécule* contient deux *Atomes*. Le premier est l'*Atome IdentityAngle* qui va recopier la valeur d'angle courante pour l'affecter comme référence pour l'asservissement. De fait, lors de la transition, le robot sera asservi sur son angle courant permettant ainsi de se prémunir d'une forte discontinuité au moment de la commutation. Le second *Atome Send0Angle* va, en étant connecté au *Produit err_pid*, permettre de réinitialiser à 0 le terme intégral du correcteur PID. Le *Graphe Moléculaire* associée à la *Molécule InitAngleControl* est présenté Figure B.9.

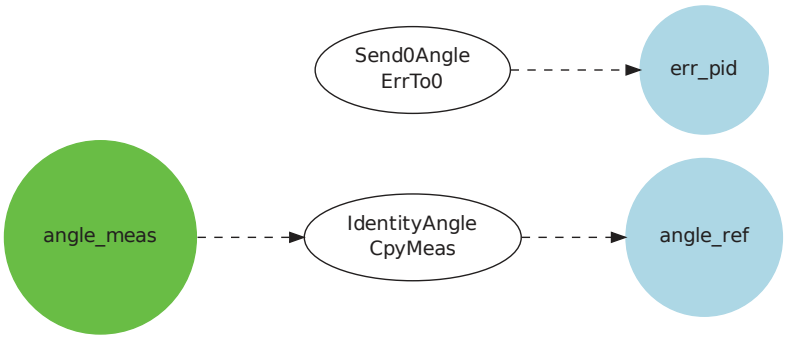

FIGURE B.9 – *Graphe Moléculaire* représentant la *Molécule InitAngleControl*

Les différents ensembles de *Liens* et *Liens d'Interface* sont définis comme :

$$L_{sel} = L(e\_sel.entityToRun, e12.entityToRun) \tag{B.190}$$

$$\begin{aligned} L_{jun} &= L(e12.angle\_ref, e2.ref\_ref) && \text{(B.191)} \\ &\quad L(e12.err\_pid, e2.err\_tt) && \text{(B.192)} \end{aligned}$$

$$\begin{aligned} l_n &= l(useControl, e\_sel.useControl) && \text{(B.193)} \\ &\quad l(angle\_meas, e12.angle\_meas) && \text{(B.194)} \\ &\quad l(angle\_meas, e2.angle\_meas) && \text{(B.195)} \\ &\quad l(spe\_meas, e2.spe\_meas) && \text{(B.196)} \\ &\quad l(t_meas, e2.t\_meas) && \text{(B.197)} \end{aligned}$$

$$\begin{aligned} l_p &= l(e1.accel, accel) && \text{(B.198)} \\ &\quad l(e2.accel, accel) && \text{(B.199)} \end{aligned}$$

Pour résumer, l'*Alternative* est schématisée Figure B.10.

Nous devons désormais vérifier la validité de cette *Alternative*. Pour cela, il faut nous baser sur les conditions établies à la Définition 41.

Il nous faut premièrement vérifier que toutes les *Entités Composables* utilisées sont valides. $e1$, $e21$ et e_sel étant des *Atomes*, ils sont par définition valides. Il reste à démontrer que les *Molécules* $e12$ et $e2$ sont valides. Cela est fait similairement au cas de l'Exemple 4.11. Ainsi, toutes les *Entités Composables* utilisées sont valides.

Nous avons $Dim(E_{jun}) = 2$. En outre, $\frac{Dim(E)!}{(Dim(E)-2)!} = \frac{2!}{0!} = 2$. Donc $Dim(E_{jun}) = \frac{Dim(E)!}{(Dim(E)-2)!}$. Il nous faut maintenant vérifier que $e1$ et $e2$ sont substituables. D'après la Définition 39, cela revient à vérifier qu'elles ont les mêmes *Produits*. Par (B.169) et (B.176), c'est bien le cas. En outre, de par (B.164), nous pouvons dire que $Pr(AngM) = Pr(e1) = Pr(e2)$.

Il ne nous reste plus qu'à vérifier la cohérence des *Liens* et *Liens d'Interface*. Similairement aux exemples précédents, nous pouvons montrer que les différents *Liens* et *Liens d'Interface* sont cohérents.

Dès lors, notre *Alternative* est valide.

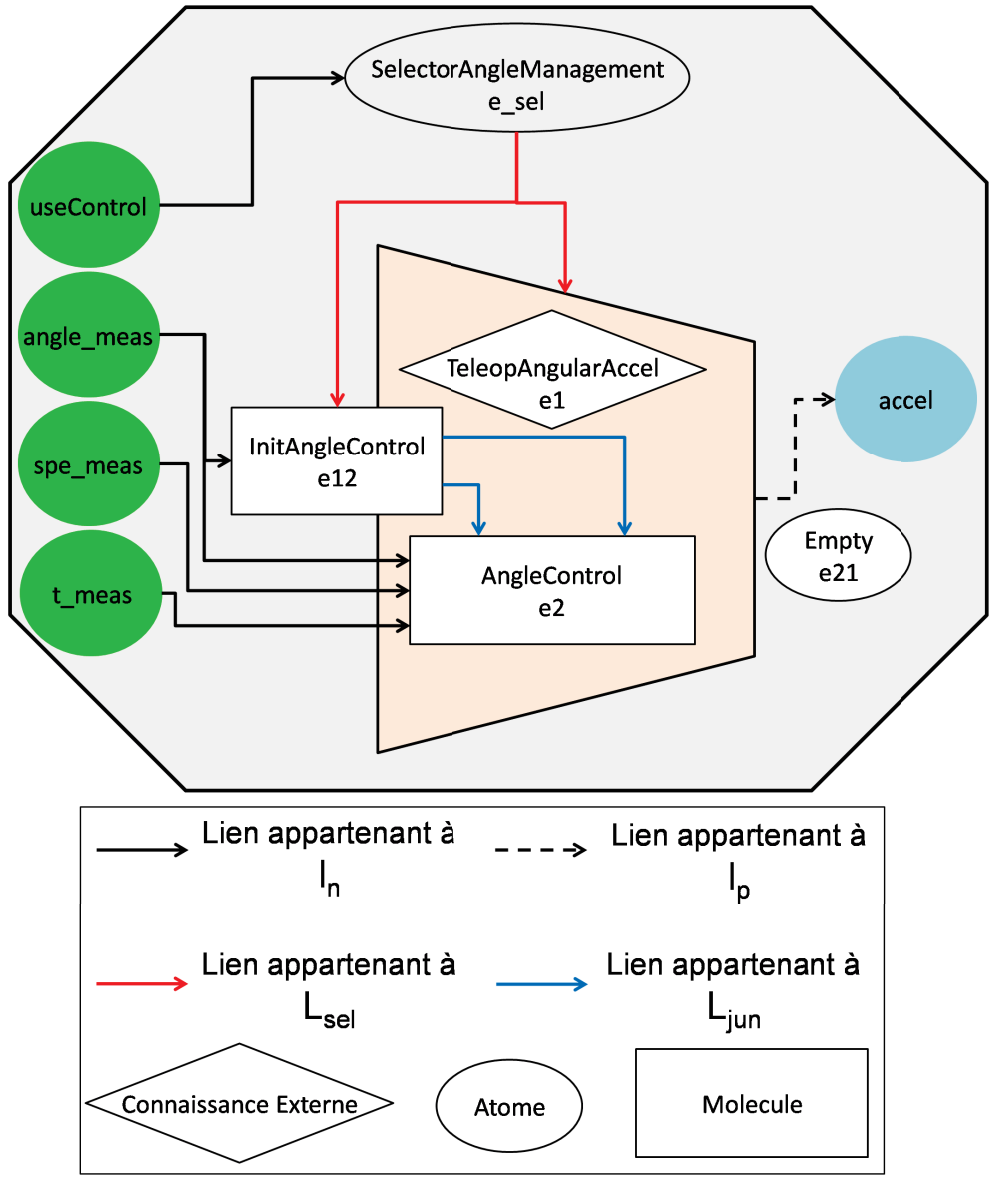

FIGURE B.10 – Représentation de l'*Alternative* AngleManagement

Annexe C

Liste des Datatypes actuellement définis

Dans cette annexe, nous allons présenter les différents *Types* définis dans nos travaux. Nous ne présenterons que les *Types* simples et complexes puisque les *Types Array* ne sont qu'une collection d'éléments d'un *Type*.

C.1 Types simples

Les *Types* simples contiennent une donnée unique. Nous allons présenter chaque *Type*, ses valeurs admissibles, les différentes unités définies. L'Unité du Système International dans laquelle est stockée la donnée dans nos objets sera repérée par l'exposant ^{USI}.

Acceleration :

Définition : Définition d'une accélération linéaire.

Valeurs admissibles : $[-\infty\ \infty]$

Unités :

- $m * s^{-2\ USI}$

Angle :

Définition : Définition d'un angle. Les radians sont choisis comme Unité du Système International puisque c'est l'unité utilisée dans tous les opérateurs trigonométriques de base.

Valeurs admissibles : $[-\infty\ \infty]$

Unités :

- $rad^{\ USI}$
- °

AngularAcceleration :

Définition : Définition d'une accélération angulaire.

Valeurs admissibles : $[-\infty\ \infty]$

Unités :

– $rad * s^{-2}$ ^{USI}

AngularDamping :

Définition : Amortissement lors d'un mouvement de rotation.

Valeurs admissibles : $[-\infty\ \infty]$

Unités :

– $N * s * rad^{-1}$ ^{USI}

AngularSpeed :

Définition : Définition d'une vitesse angulaire.

Valeurs admissibles : $[-\infty\ \infty]$

Unités :

– $rad * s^{-1}$ ^{USI}

Boolean :

Définition : Définition d'un booléen.

Valeurs admissibles : $\{true, false\}$

Unités :

– $sans\ unité$ ^{USI}

Damping :

Définition : Amortissement lors d'un mouvement linéaire.

Valeurs admissibles : $[-\infty\ \infty]$

Unités :

– $N * s * m^{-1}$ ^{USI}

Distance :

Définition : Distance entre deux points de l'espace.

Valeurs admissibles : [0 ∞]

Unités :

- m ^{USI}
- cm
- km

Duration :

Définition : Définition d'une durée.

Valeurs admissibles : [0 ∞]

Unités :

- s ^{USI}

Float :

Définition : Un nombre à virgule sans unité.

Valeurs admissibles : [$-\infty$ ∞]

Unités :

- $sans\ unité$ ^{USI}

Force :

Définition : Définition d'une force.

Valeurs admissibles : [$-\infty$ ∞]

Unités :

- N ^{USI}
- mN ^{USI}

Frequency :

Définition : Définition d'une fréquence.

Valeurs admissibles : [0 ∞]

Unités :

- Hz ^{USI}

FrequencySquared :

Définition : Définition d'une fréquence mise au carré.

Valeurs admissibles : [0 ∞]

Unités :

– $Hz^{2\ USI}$

FrequencyCubed :

Définition : Définition d'une fréquence à la puissance 3.

Valeurs admissibles : [0 ∞]

Unités :

– $Hz^{3\ USI}$

Inertia :

Définition : Définition d'une inertie.

Valeurs admissibles : [0 ∞]

Unités :

– $Kg * m^{2\ USI}$

Integer :

Définition : Un nombre entier sans unité.

Valeurs admissibles : [$-\infty$ ∞]

Unités :

– $sans\ unité^{\ USI}$

InverseAngle :

Définition : L'inverse d'un angle.

Valeurs admissibles : [$-\infty$ ∞]

Unités :

– $rad^{-1\ USI}$

InverseForce :

Définition : L'inverse d'une force.

Valeurs admissibles : $[-\infty\ \infty]$

Unités :

- N^{-1} ^{USI}

Mass :

Définition : Une masse.

Valeurs admissibles : $[0\ \infty]$

Unités :

- Kg ^{USI}
- g

Position :

Définition : Définition d'une position.

Valeurs admissibles : $[-\infty\ \infty]$

Unités :

- m ^{USI}
- cm
- km

SignedInertia :

Définition : Une inertie qui peut être négative (nécessaire pour le modèle dynamique du robot).

Valeurs admissibles : $[-\infty\ \infty]$

Unités :

- $Kg * m^2$ ^{USI}

SignedMass :

Définition : Une masse qui peut être négative (nécessaire pour le modèle dynamique du robot).

Valeurs admissibles : $[-\infty\ \infty]$

Unités :

- $Kg^{\ USI}$
- g

Speed :

Définition : Une vitesse linéaire.

Valeurs admissibles : $[-\infty\ \infty]$

Unités :

- $m * s^{-1\ USI}$

SquareAngularDamping :

Définition : Amortissement angulaire élevé au carré.

Valeurs admissibles : $[-\infty\ \infty]$

Unités :

- $N^2 * s^2 * rad^{-2\ USI}$

SquareDamping :

Définition : Amortissement linéaire élevé au carré.

Valeurs admissibles : $[-\infty\ \infty]$

Unités :

- $N^2 * s^2 * m^{-2\ USI}$

Stiffness :

Définition : Raideur d'un ressort.

Valeurs admissibles : $[-\infty\ \infty]$

Unités :

- $N * m^{-1\ USI}$

TextString :

Définition : Définition d'une chaîne de caractères.

Valeurs admissibles : Toute chaîne de caractères

Unités :

– *sans unité* ^{USI}

Torque :

Définition : Définition d'un couple.

Valeurs admissibles : $[-\infty\ \infty]$

Unités :

– $N * m$ ^{USI}

UInteger :

Définition : Définition d'un nombre entier positif sans unité.

Valeurs admissibles : $[0\ \infty]$

Unités :

– *sans unité* ^{USI}

C.2 Types complexes

Les *Types* complexes sont des concaténations de *Types* simples. Dans notre cas, nous avons définis deux types qui représentent des systèmes de coordonnées. De fait, ils n'ont pas de champ *Axis*.

CartesianPose :

Contenu :

– Position X
– Position Y
– Position Z
– Angle phi
– Angle theta
– Angle psi

Définition : CartesianPose sert à représenter les coordonnées d'un point dans un repère cartésien et les orientations associées. Cela permet par exemple de contenir la position de notre robot. Ce *Type* permet également de représenter les coordonnées et orientations d'un repère par rapport à un autre.

CylindricalPoint :

Contenu :

- Distance r
- Angle theta
- Distance h
- Angle or_y
- Angle or_z

Définition : CylindricalPoint sert à représenter les coordonnées d'un point dans un repère cylindrique. En outre, nous utilisons deux données supplémentaires, *or_y* et *or_z*, pour contrôler l'orientation de notre cylindre par rapport à sa configuration normale (base dans le plan xy et axe principal suivant z). Ceci est illustré sur la Figure C.1.

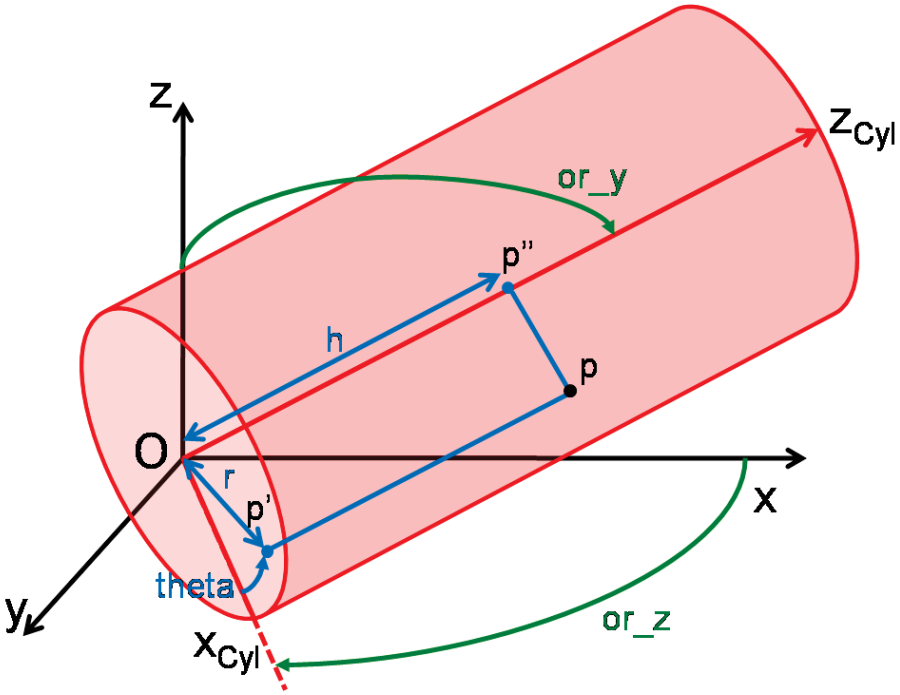

FIGURE C.1 – Signification des différentes données comprises dans le *Type* CylindricalPoint

Soit un repère cartésien d'origine O et d'axes $\vec{Ox}$, $\vec{Oy}$, $\vec{Oz}$. Cyl est le cylindre définissant le repère de coordonnées cylindriques, il a pour origine O. La base du cylindre est définie par les vecteurs $\vec{Ox_{Cyl}}$, $\vec{Oy_{Cyl}}$ même si ce dernier vecteur n'est pas représenté dans notre Figure

afin d'alléger celle-ci. L'axe du Cylindre est donné par le vecteur $\vec{Oz_{Cyl}}$.

Soit p un point. p' est sa projection sur la base de Cyl. r est définie comme la norme du vecteur $\vec{Op'}$. $theta$ est l'angle entre le vecteur $\vec{Ox}$ et le vecteur $\vec{Op'}$. Soit p'' la projection de p sur l'axe de Cyl. h est définie comme la distance $[Op'']$.

Enfin, comme dit précédemment, nous pouvons contrôler l'orientation de Cyl par rapport au repère cartésien. En effet, traditionnellement, la base du cylindre correspond au plan $\vec{Ox}$, $\vec{Oy}$ et son axe est défini comme le vecteur $\vec{Oz}$. Ici, nous pouvons faire tourner la base du cylindre autour de l'axe $\vec{Oz}$. Ainsi, or_z définit l'angle entre $\vec{Ox}$ et $\vec{Ox_{Cyl}}$. Nous pouvons également orienter l'axe du cylindre en faisant tourner celui-ci autour de l'axe $\vec{Oy}$. Ainsi, or_y est l'angle entre $\vec{Oz}$ et $\vec{Oz_{Cyl}}$.

Annexe D

Implémentation des concepts associés aux Entités Composables

Aux Chapitre 4 et au Chapitre 5, nous avons présenté les différents concepts et entités que nous avons développés. A partir de ces concepts, notre approche propose une méthodologie permettant de guider l'utilisation des connaissances par le logiciel de contrôle. Pour pouvoir être utilisée par le logiciel de contrôle, nos connaissances doivent être mises en œuvre dans des entités logicielles (Figure D.1). Ces dernières doivent traduire le plus fidèlement possible les concepts proposés dans notre approche. En effet, des propriétés telles que la vérification de la cohérence des *Liens* sont cruciales pour limiter les risques d'erreurs lors du développement du logiciel de contrôle.

Notre approche étant basée sur le paradigme objet, nous avons choisi de développer une API C++ qui servira de base à la conception de nos entités logicielles. Nous présenterons ici cette API ainsi que les *Entités Composables* implémentables. De plus, nous souhaitons évidemment posséder une description générique de nos *Entités Composables*. Nous présenterons celle réalisée ainsi que ses limitations actuelles. Enfin, nous introduirons l'outil de génération de code utilisé afin de simplifier le développement des entités logicielles depuis la description générique tout en réduisant les risques d'erreur lors du processus.

D.1 API proposée et entités implémentables

D.1.1 Présentation de l'API

Pour implémenter nos *Entités Composables* sous forme d'entités logicielles, nous proposons une Interface de Programmation (API) qui va définir la structure commune à ces entités logicielles. La première préoccupation est de choisir le langage dans lequel sera proposée cette API.

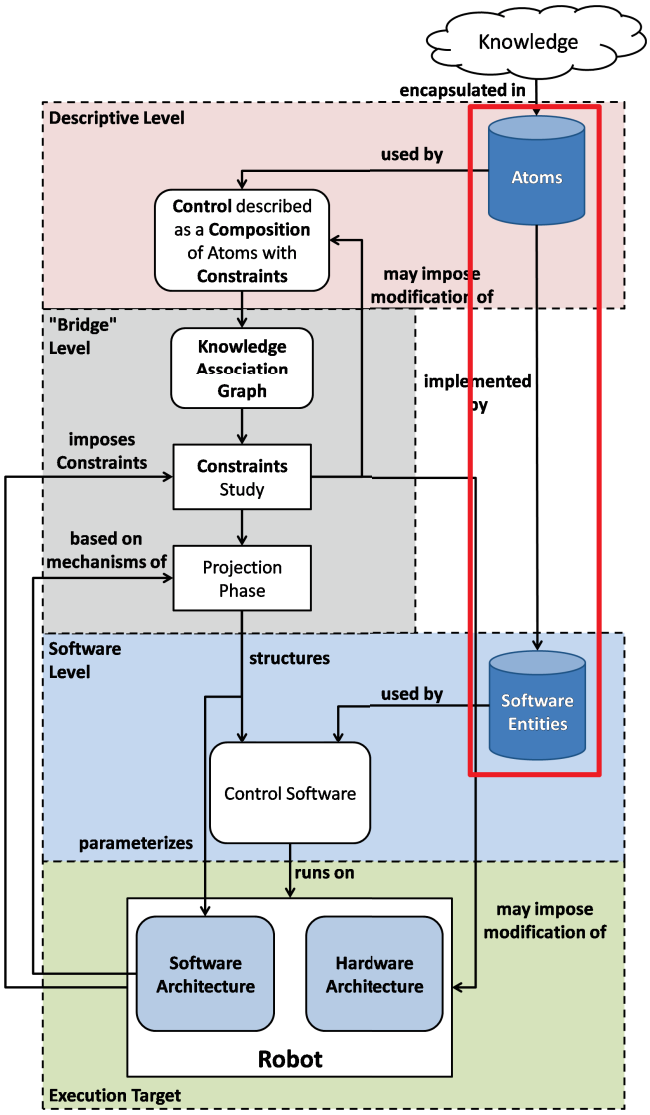

FIGURE D.1 – Aspects de la méthodologie abordés dans l'Annexe D

Pour cela, la première contrainte à prendre en considération est celle des langages supportés par le *Middleware* temps-réel ContrACT qui est utilisé pour notre projection. Celui-ci permet d'utiliser deux langages le C et le C++. Notre approche étant inspirée du paradigme objet, nous avons de fait choisi de développer une API C++, afin que l'implémentation réalisée soit aussi fidèle que possible à nos concepts théoriques.

La Figure D.2 présente le diagramme UML des différentes classes et énumérations utilisées.

Les énumérations $frame$ and $axis$ servent à définir l'ensemble des valeurs possibles pour les champs $Frame$ et $Axis$.

La classe $abstractDatatype$ sert à définir la structure de base de toutes les classes définissant les *Datatypes*. Elle sert aussi de classe de base générique utilisée pour mettre en place le polymorphisme dans la définition de l'interface de nos entités logicielles.

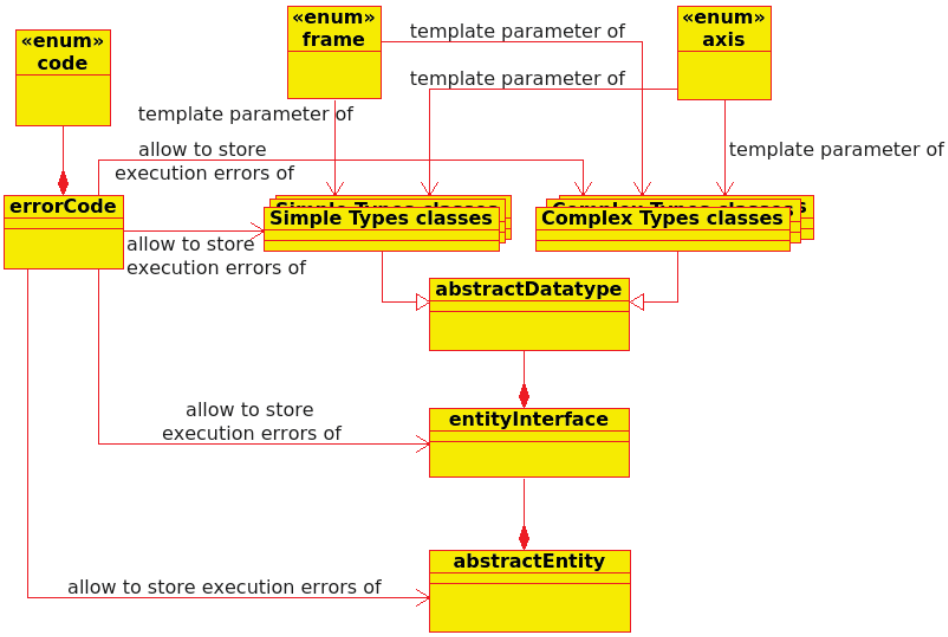

FIGURE D.2 – Diagramme UML de l'API utilisée

Les classes définissant les différents *Datatypes* héritent toutes d'*abstractDatatype*. Le *Type* détermine le nom de la classe et les champs *Frame* et *Axis* sont représentés par des paramètres template de la classe comme le montre l'Exemple de Code D.1.

Exemple de Code D.1 – Définition d'une classe Datatype : la classe Distance

```
typedef double distance;
template <frame F, axis A>
class Distance : public abstractDatatype
{
public:
   Distance();
   Distance(distance dist);
   virtual ~Distance(void);

   virtual frame getDataFrame() const;
   virtual axis getDataAxis() const;

   common::Distance<F,A>& operator=(const common::Distance<F,A>& b);

   bool operator==(const common::Distance<F,A>& b) const;

   errorCode fromUSI(distance D_USI);
```

```
18    distance toUSI() const;
19
20    errorCode fromMeter(distance D_m);
21    distance toMeter() const;
22
23    errorCode fromKilometer(distance D_km);
24    distance toKilometer() const;
25
26    errorCode fromCentimeter(distance D_cm);
27    distance toCentimeter() const;
28
29 private:
30    distance m_Distance; //Distance in meters
31 };
```

La liste complète des *Types* définis est disponible à l'Annexe C.

L'utilisation de paramètres *template* pour représenter les champs $Frame$ et $Axis$ s'explique par la volonté de permettre la vérification de la cohérence des *Liens* (Définition 27) entre entités au moment de la compilation. En effet, lorsque nous instancions une classe *Datatype*, nous fixons les valeurs de ses champs $Frame$ et $Axis$. Le compilateur va ensuite vérifier que les classes connectées sont identiques (i.e. même nom de classe et aussi mêmes valeurs de paramètres *template*) au moment de la compilation. Si les paramètres *template* des types sont différents (i.e. le *Lien* est incohérent), la liaison sera rejetée et le programme ne compilera pas. Cela nous permet d'éliminer un grand nombre d'erreurs avant l'exécution du programme. Cette propriété est intéressante car la réponse adéquate à apporter, durant l'exécution, à de telles erreurs qui rendent inutilisables certaines portions du logiciel de contrôle, peut se révéler très complexe à déterminer et à réaliser en ligne dans une application temps-réel.

Il faut néanmoins noter que si la connexion porte sur des *Types Array*, il n'est alors pas possible de complètement vérifier la cohérence du *Lien* puisque la taille des tableaux est, dans certains cas, dynamique. L'implémentation des *Types Array* ne peut dès lors considérer que les tableaux ont une taille fixe. L'égalité des tailles ne pourra donc être vérifiée qu'à l'exécution.

De plus, dans l'Exemple de Code D.1, comme nous l'avons dit au Chapitre 4, la classe encapsule la donnée, stockée dans l'unité du Système International (USI) associée à la grandeur physique, et permet à l'utilisateur un accès dans les différentes unités définies (liste bien entendu extensible au besoin). Les accesseurs en écriture renvoient tous un *errorCode*.

La classe *errorCode* nous permet de remonter les différentes erreurs qui peuvent se produire dans l'utilisation de nos entités logicielles, lors de l'exécution de leur *Physique* ou lors de la manipulation des *Datatypes*. C'est également via cette classe qu'est indiqué si les tailles de deux *Types Array* ne concordent pas. L'Exemple de Code D.2 présente la définition de cette classe.

Exemple de Code D.2 – Définition de la classe errorCode

```
class errorCode
{
public:
   errorCode();
   errorCode(const errorCode& e);
   ~errorCode();

   errorCode& operator+=(const errorCode& e); // Addition of a list of errors to
       the list
   errorCode& operator+=(const code& e); // Addition of a single error to the list

   bool noError() const; // Checking if error list is empty
   std::vector<code> get_errors() const; // Returns error list

   void displayError(std::string text, bool displayOK = false) const; // Display
       errors

protected:
   std::vector<code> m_err;
};
```

Les erreurs sont stockées sous la forme d'une liste chronologique (i.e. la première erreur de la liste est la première à s'être produite). L'énumération *code* (Exemple de Code D.3) contient l'ensemble des codes d'erreurs définis jusqu'à présent.

Exemple de Code D.3 – Liste des codes d'erreur définis

```
enum code
{
   // Basic error defines
    ERR_UNKNOWN = 1, // The execution failed for an unspecified reason
    ERR_FILE_OPEN = 2, // A file could not be opened
```

```
6     ERR_OUT_OF_ARRAY = 3, //An index was out of the array range
7     ERR_NO_SIZE_MATCH = 4, // The size of 2 arrays don't match each other
8
9    // Error in entity links
10    ERR_AFFECT_FRAME = 50, // No frame correspondance in an affectation
11    ERR_AFFECT_AXIS = 51, // No axis correspondance
12    ERR_AFFECT_TYPE = 52, // No type correspondance (i.e. affectation doesn't fail
          because of frame or axis problem)
13    ERR_PORT_NAME = 53, // Trying to access a port with a name that does not exist
14
15   // Error in units
16    ERR_NEG_MASS = 100, //Error in trying to manipulate a negative mass
17    ERR_NEG_DISTANCE = 101, //Error in trying to manipulate a negative distance
18    ERR_NEG_INERTIA = 102, //Error in trying to manipulate a negative inertia
19    ERR_NEG_DURATION = 103, //Error in trying to manipulate a negative duration
20    ERR_NEG_FREQUENCY = 104, //Error in trying to manipulate a negative frequency
21    ERR_NEG_FREQUENCYSQUARED = 105, //Error in trying to manipulate a negative
          frequency squared
22    ERR_NEG_FREQUENCYCUBED = 106 //Error in trying to manipulate a negative
          frequency cubed
23 };
```

L'accesseur *fromCentimeter* (Exemple de Code D.4) de la classe *Distance* (Exemple de Code D.1) illustre la manière d'utiliser cette classe pour récupérer puis transmettre les erreurs.

Exemple de Code D.4 – Accesseur fromCentimeter de la classe Distance

```
1 template <frame F, axis A>
2 errorCode Distance<F,A>::fromCentimeter(distance D_cm)
3 {
4    errorCode err; // Empty error list
5    if(D_cm >= 0) // A distance is always positive
6    {
7       m_Distance = D_cm / 100; // Conversion to USI
8       return err; // No error, return an empty list
9    }
10   else
11   {
12      return err + ERR_NEG_DISTANCE; // Add Error to the list
```

```
13    }
14 }
```

L'utilisation de ces codes d'erreur est utile à la fois pour debugger l'application et également lors de l'exécution pour faire remonter au système décisionnel (opérateur humain et/ou superviseur logiciel) toutes les erreurs affectant le logiciel de contrôle afin qu'une réponse puisse être trouvée si possible.

La classe *entityInterface* définit la structure commune aux trois ensembles (*Besoins*, *Produits* et *Stockages Internes*) qui définissent l'interface de nos entités. Sa définition est donnée dans l'Exemple de Code D.5.

Exemple de Code D.5 – Définition de la classe entityInterface

```
1  class entityInterface : public std::map<std::string, abstractDatatype *>
2  {
3  public :
4     entityInterface();
5
6     template<typename U>
7     U* getData(std::string name, errorCode* e = NULL) const;
8
9     template<typename U>
10    errorCode setData(std::string name, const U& data);
11
12    template<typename U>
13    void addPort(std::string name, U* data);
14 };
```

Cette classe dérive de la classe *map* de la bibliothèque standard du C++ et définit chaque *Elément d'Interface* comme un couple $(Nom, Datatype)$, conformément à la Définition 6. La méthode *addPort* permet d'ajouter un élément à l'interface. *setData* permet de modifier la valeur d'un élément existant et *getData* permet quant à elle d'accéder à la valeur d'un élément existant.

Enfin, la classe *abstractEntity* définit la structure de base et les fonctionnalités communes à nos entités logicielles. Elle définit les trois ensembles d'*Eléments d'Interface* et la fonction contenant la *Physique* qui devra être redéfinie spécifiquement pour chaque applica-

tion (Exemple de Code D.6, ligne 7). C'est en outre dans le destructeur de cette classe que s'effectue la libération de la mémoire occupée par les *Eléments d'Interface*.

Exemple de Code D.6 – Définition de la classe abstractEntity

```
1  class abstractEntity
2  {
3  public :
4     abstractEntity();
5     ~abstractEntity();
6
7     virtual errorCode physics() = 0; //Description of your component physics. This
          method must be redefined specifically for each component
8
9  protected:
10    entityInterface m_products;
11    entityInterface m_needs;
12    entityInterface m_internalStorages;
13 };
```

Nos *Entités Composables* seront implémentées comme des classes héritant d'*abstractEntity*.

D.1.2 Entités implémentables

Evidemment toutes les *Entités Composables* que nous avons définies ne sont pas implémentées avec cette API car cela dépend de leur rôle. En effet, seules les entités effectuant des opérations calculatoires ou de fixation de paramètres (initialisations) sont à implémenter. Celles dont le rôle est structurel, qui peuvent soit être traduites via des mécanismes de l'architecture logicielle soit disparaître lors de la phase d'implémentation, ne le sont pas.

Ainsi les *Atomes* (hors *Connaissances Externes*), qui tombent dans la première catégorie, sont implémentables via cette API. A contrario, les *Alternatives* permettent de structurer les commutations et ne seront donc pas mises en œuvre de cette manière.

Les *Connaissances Externes* servent à représenter des connaissances spécifiques à la cible d'implémentation. Cela inclut notamment les *drivers* des différents capteurs, actionneurs ou média de communication utilisés ainsi que les fonctions de lecture et d'écriture dans des fichiers. La manière de les implémenter est laissée à l'appréciation des utilisateurs. Néanmoins, il est possible de donner certains conseils pratiques. Il n'est en général ni utile ni pertinent d'encapsuler les *drivers* du matériel dans nos entités logicielles. En effet, cela peut complexifier la mise

en œuvre de leurs différentes fonctionnalités. Par contre, suivant les mécanismes de transfert de données offerts par l'architecture logicielle, il peut être intéressant de développer une entité logicielle chargée de convertir les données brutes sortant du *driver* en *Datatype*. Mais, avec notre API, cela n'est valable que si le *Middleware* permet l'échange d'objets. D'un autre côté, des fonctionnalités telles que la lecture ou l'écriture dans des fichiers (la lecture d'un fichier est généralement la manière privilégiée pour mettre en œuvre les *Atomes* d'*Expertise*), dont l'implémentation est généralement plutôt simple, peuvent être encapsulées dans des entités logicielles développées avec notre API. Cela est d'autant plus pertinent que ces fonctionnalités sont souvent utilisées en connexion directe avec d'autres entités logicielles que nous mettons en œuvre.

Enfin, le cas des *Molécules* doit être considéré avec plus d'attention. En effet, celles-ci permettent principalement une meilleure structuration des *Entités Composables* utilisées dans notre description de connaissances. Mais elles facilitent également la réutilisation d'un ensemble d'*Entités Composables* d'une description à l'autre. A l'instar de cet apport, il peut être intéressant de pouvoir réutiliser un groupe d'entités logicielles qui sont couramment associées. Cela éviterait de devoir refaire les liaisons à chaque utilisation de ces entités ce qui faciliterait leur manipulation tout en réduisant les risques d'erreurs. Pour pouvoir alors déterminer quelles *Molécules* jouent ce rôle, il nous faut préciser la notion de *Molécule*.

Définition 47:

Une *Molécule* m est dite simple, noté $m \in MS$, avec MS l'ensemble des *Molécules simples*, si et seulement si, notant $Phy(m) = (E, L, l_n, l_p)$:
$E_i \in (At{-}EK) \cup (MS{-}\{m\}) \forall i \in [1..Dim(E)]$ (Une *Molécule simple* est une *Molécule* ne contenant que des *Atomes*, hors *Connaissances Externes*, ou d'autres *Molécules simples*.)
m est acyclique c'est-à-dire que $\forall \{L_i, .., L_j\} \subset L$ l'ensemble des entités liées par ces liens $\{e_x, .., e_y\}$ est tel que $e_x \neq e_y$ (Il n'y a pas de cycle au sein de la *Molécule*.)

Ainsi, les *Molécules simples* pourront être implémentées avec notre API. La contrainte d'acyclicité provient de notre méthode d'ordonnancement des différentes entités au sein de la *Molécule*, comme expliqué Section D.2.5.

D.1.3 Exemples d'utilisation de l'API

Nous allons maintenant illustrer à travers différents exemples comment implémenter *Atomes* et *Molécules simples* avec notre API.

Exemple D.1:

Considérons l'*Atome* présenté à l'Exemple A.2, la définition de la classe l'implémentant est

donnée dans l'Exemple de Code D.7.

Exemple de Code D.7 – Définition de la classe ResultingForceComputation

```
namespace Environment
{
   namespace Karst
   {
      template <common::frame Ent, common::axis Ax1, common::axis Ax2>
      class ResultingForceComputation : public common::abstractEntity
      {
      public :
         ResultingForceComputation();
         virtual ~ResultingForceComputation();
         virtual common::errorCode physics();

         common::errorCode set_need_Npts(const common::UInteger<common::NOFRAME,
             common::NOAXIS>& Npts);
         common::errorCode set_need_KDVZ(const common::Stiffness<common::NOFRAME,
             common::NOAXIS>& KDVZ);
         common::errorCode set_need_intru(const common::Array<
             common::CylindricPoint<Ent> >& intru);

         common::Force<Ent, Ax1> get_product_F1(common::errorCode* e = NULL) const;
         common::Force<Ent, Ax2> get_product_F2(common::errorCode* e = NULL) const;

      protected :
         common::UInteger<common::NOFRAME, common::NOAXIS>
             get_need_Npts(common::errorCode* e = NULL) const;
         common::Stiffness<common::NOFRAME, common::NOAXIS>
             get_need_KDVZ(common::errorCode* e = NULL) const;
         common::Array< common::CylindricPoint<Ent> >
             get_need_intru(common::errorCode* e = NULL) const;

         common::errorCode set_product_F1(const common::Force<Ent, Ax1>& F1);
         common::errorCode set_product_F2(const common::Force<Ent, Ax2>& F2);

      };
   }
```

```
30 }
31
32 #include "ResultingForceComputation.aid"
```

Cet exemple illustre comment nos différents concepts sont implémentés. Les lignes 1 et 3 montrent que les *Domaines de Connaissance* sont représentés par une hiérarchie de *namespaces*. Ce choix nous permet à la fois d'indiquer la provenance des connaissances mises en œuvre tout en n'affectant pas, comme nous l'avons souhaité, l'utilisation faite de la connaissance.

L'*Atome* en lui-même est, comme nous l'avons dit précédemment, implémenté comme une classe héritant de *abstractEntity* (ligne 6). Ses *Paramètres d'Interface* sont implémentés sous forme de paramètres *template* de notre classe qui servent à valuer les paramètres *template* des *Eléments d'Interface* (ligne 17 et 18 par exemple). Cette classe doit obligatoirement implémenter la méthode *physics* purement virtuelle de la classe *abstractEntity* afin de décrire la *Physique* propre à l'*Atome* (ligne 11).

Nous implémentons également les accesseurs à nos différents éléments. Ceux-ci sont toujours nommés selon la convention suivante :

$$Acces\_Type\_Nom$$

Acces indique le type d'accès à l'élément, *set* indique un accès en écriture et *get* un accès en lecture. *Type* nous permet de signifier s'il s'agit d'un *Besoin* (*need*), d'un *Produit* (*product*) ou d'un *Stockage Interne* (*storage*). Enfin, *Nom* est le nom du *Paramètre d'Interface*.

Dans notre cas, les accesseurs en écriture des *Besoins* et les accesseurs en lecture des *Produits* seront toujours *public*, c'est-à-dire accessibles par d'autres entités logicielles. Les accesseurs en lecture des *Besoins* et en écriture des *Produits* seront toujours *protected*, c'est-à-dire accessibles seulement par la *Physique*. Enfin, les accesseurs des *Stockages Internes* seront toujours *public* car, dans le cadre de l'implémentation des *Alternatives*, les entités chargées d'effectuer la jonction auront besoin d'y accéder. D'autres mécanismes tels que l'amitié [1] peuvent donner cet accès tout en empêchant la manipulation des *Stockages Internes* par des entités logicielles non autorisées. Mais ils sont trop contraignants à mettre en œuvre car ils nécessitent de modifier le code de notre entité logicielle à chaque implémentation.

Enfin, si notre entité comporte des *Paramètres d'Interface*, on ne peut alors plus implémenter le code de nos méthodes dans un fichier source (d'extension *.cpp*). Néanmoins, pour garder la séparation définition/implémentation qui est indispensable pour conserver une bonne lisibilité du code produit, nous avons choisi de stocker l'implémentation dans un fichier d'ex-

1. `http://en.cppreference.com/w/cpp/language/friend`

tension *.aid* (pour *Atom Implementation Detail*). Celui-ci est inclus dans notre fichier d'en-tête (ligne 32) et, dès lors, le compilateur considèrera le contenu du *.aid* comme faisant partie du fichier d'en-tête. Par contre si l'*Entité Composable* n'a pas de *Paramètres d'Interface*, et donc la classe n'a pas de paramètres *template*, on privilégiera l'écriture du code dans un fichier source. En effet, cela permettra de compiler l'entité logicielle dans la bibliothèque et pas au moment de la compilation de l'application comme c'est le cas si l'implémentation est écrite dans le fichier en-tête. Comme notre calculateur, la BeagleBone, n'a que peu de ressources calculatoires disponibles, avoir des entités déjà compilées permet de réduire significativement la durée de compilation de l'application.

Voyons maintenant l'implémentation des différentes méthodes de notre classe.

Exemple de Code D.8 – Constructeur et Destructeur de la classe ResultingForceComputation

```
1  template <common::frame Ent, common::axis Ax1, common::axis Ax2>
2  ResultingForceComputation<Ent, Ax1, Ax2>::ResultingForceComputation() :
       common::abstractEntity()
3  {
4     m_needs.addPort("Npts", new common::UInteger<common::NOFRAME, common::NOAXIS>);
5     m_needs.addPort("KDVZ", new common::Stiffness<common::NOFRAME, common::NOAXIS>);
6     m_needs.addPort("intru", new common::Array< common::CylindricPoint<Ent> >(1));
7
8     m_products.addPort("F1", new common::Force<Ent, Ax1>);
9     m_products.addPort("F2", new common::Force<Ent, Ax2>);
10
11 }
12
13 template <common::frame Ent, common::axis Ax1, common::axis Ax2>
14 ResultingForceComputation<Ent, Ax1, Ax2>::~ResultingForceComputation()
15 {
16 }
```

L'Exemple de Code D.8 présente le constructeur et le destructeur de la classe. Le constructeur sert à déclarer les différents *Eléments d'Interface* de notre entité. Le destructeur ne fait rien car, comme nous l'avons vu précédemment, c'est dans le destructeur de *abstractEntity* qu'est assurée la libération de l'espace mémoire occupé par l'*Interface* de notre entité. Par contre, si le code interne nécessite d'utiliser des opérations spécifiques à la fin de l'exécution, c'est dans cette méthode qu'elles devront être codées.

L'Exemple de Code D.9 présente le code des accesseurs en lecture et écriture des *Besoins*.

Exemple de Code D.9 – Accesseurs des Besoins de la classe ResultingForceComputation

```
template <common::frame Ent, common::axis Ax1, common::axis Ax2>
common::errorCode ResultingForceComputation<Ent, Ax1, Ax2>::set_need_Npts(const
    common::UInteger<common::NOFRAME, common::NOAXIS>& Npts)
{
   common::errorCode err = m_needs.setData("Npts", Npts);
   if(err.noError()) // Array size Modification
   {
      common::Array< common::CylindricPoint<Ent> >* out = m_needs.getData<
          common::Array< common::CylindricPoint<Ent> > >("intru", &err);
      err += out->set_size(get_need_Npts(&err).toNoUnit());
   }
   return err;
}

template <common::frame Ent, common::axis Ax1, common::axis Ax2>
common::errorCode ResultingForceComputation<Ent, Ax1, Ax2>::set_need_KDVZ(const
    common::Stiffness<common::NOFRAME, common::NOAXIS>& KDVZ)
{
   return m_needs.setData("KDVZ", KDVZ);
}

template <common::frame Ent, common::axis Ax1, common::axis Ax2>
common::errorCode ResultingForceComputation<Ent, Ax1, Ax2>::set_need_intru(const
    common::Array< common::CylindricPoint<Ent> >& intru)
{
   common::errorCode err;
   common::Array< common::CylindricPoint<Ent> >*prec = m_needs.getData<
       common::Array< common::CylindricPoint<Ent> > >("intru", &err);
   if(!err.noError())
   {
      return err;
   }
   else if(prec->get_size() != intru.get_size()) // Checking if Array size matches
   {
      return (err += ERR_NO_SIZE_MATCH);
```

```
31    }
32    else
33    {
34       return m_needs.setData("intru", intru);
35    }
36 }
37 ////////////////////////////////////////////////////////////////////////////////
38 template <common::frame Ent, common::axis Ax1, common::axis Ax2>
39 common::UInteger<common::NOFRAME, common::NOAXIS> ResultingForceComputation<Ent,
       Ax1, Ax2>::get_need_Npts(common::errorCode* e) const
40 {
41    common::UInteger<common::NOFRAME, common::NOAXIS>* out = m_needs.getData<
          common::UInteger<common::NOFRAME, common::NOAXIS> >("Npts", e);
42    if(out != NULL)
43    {
44       return *out;
45    }
46    else
47    {
48       return common::UInteger<common::NOFRAME, common::NOAXIS>();
49    }
50 }
51
52 template <common::frame Ent, common::axis Ax1, common::axis Ax2>
53 common::Stiffness<common::NOFRAME, common::NOAXIS> ResultingForceComputation<Ent,
       Ax1, Ax2>::get_need_KDVZ(common::errorCode* e) const
54 {
55    common::Stiffness<common::NOFRAME, common::NOAXIS>* out = m_needs.getData<
          common::Stiffness<common::NOFRAME, common::NOAXIS> >("KDVZ", e);
56    if(out != NULL)
57    {
58       return *out;
59    }
60    else
61    {
62       return common::Stiffness<common::NOFRAME, common::NOAXIS>();
63    }
```

```
64 }
65
66 template <common::frame Ent, common::axis Ax1, common::axis Ax2>
67 common::Array< common::CylindricPoint<Ent> > ResultingForceComputation<Ent, Ax1,
       Ax2>::get_need_intru(common::errorCode* e) const
68 {
69    common::Array< common::CylindricPoint<Ent> >* out = m_needs.getData<
          common::Array< common::CylindricPoint<Ent> > >("intru", e);
70    if(out != NULL)
71    {
72       return *out;
73    }
74    else
75    {
76       return common::Array< common::CylindricPoint<Ent> >(1);
77    }
78 }
```

Les méthodes d'accès en écriture se basent sur la fonction *setData* pour mettre à jour la valeur du paramètre d'interface (lignes 14 à 17). Néanmoins, lorsqu'un élément de *Type Array* est présent dans l'*Interface*, cela entraîne l'ajout de quelques contraintes supplémentaires. Premièrement, l'*Elément d'Interface* contenant la taille du tableau (dans notre cas $Npts$) doit modifier la taille de celui-ci en utilisant la fonctionnalité dédiée (set_size) dans la classe *Array* (lignes 5 à 9). La méthode permettant l'accès en écriture au tableau doit quant à elle vérifier la bonne cohérence des tailles entre le tableau courant et la nouvelle valeur à lui affecter (lignes 20 à 36).

Les accès en lecture se basent sur la méthode *getData*. Il est à noter qu'en cas d'erreur d'accès, un objet *Datatype* avec une valeur par défaut est renvoyé. Dans le cas d'un tableau, sa taille, obligatoire à indiquer, est fixée à 1 par défaut.

L'Exemple de Code D.10 illustre que le codage des accesseurs des *Produits* est identique en principe à celui des *Besoins*. Seul l'ensemble d'*Eléments d'Interface* considéré est modifié. Il en va de même pour les accesseurs des *Stockages Internes* qui ne seront pas présentés dans nos exemples.

Exemple de Code D.10 – Accesseurs des Produits de la classe ResultingForceComputation

```
1 template <common::frame Ent, common::axis Ax1, common::axis Ax2>
2 common::Force<Ent, Ax1> ResultingForceComputation<Ent, Ax1,
```

```
    Ax2>::get_product_F1(common::errorCode* e) const
{
   common::Force<Ent, Ax1>* out = m_products.getData< common::Force<Ent, Ax1>
       >("F1", e);
   if(out != NULL)
   {
      return *out;
   }
   else
   {
      return common::Force<Ent, Ax1>();
   }
}

template <common::frame Ent, common::axis Ax1, common::axis Ax2>
common::Force<Ent, Ax2> ResultingForceComputation<Ent, Ax1,
    Ax2>::get_product_F2(common::errorCode* e) const
{
   common::Force<Ent, Ax2>* out = m_products.getData< common::Force<Ent, Ax2>
       >("F2", e);
   if(out != NULL)
   {
      return *out;
   }
   else
   {
      return common::Force<Ent, Ax2>();
   }
}
////////////////////////////////////////////////////////////////////////////////
template <common::frame Ent, common::axis Ax1, common::axis Ax2>
common::errorCode ResultingForceComputation<Ent, Ax1, Ax2>::set_product_F1(const
    common::Force<Ent, Ax1>& F1)
{
   return m_products.setData("F1", F1);
}

```

```
35 template <common::frame Ent, common::axis Ax1, common::axis Ax2>
36 common::errorCode ResultingForceComputation<Ent, Ax1, Ax2>::set_product_F2(const
       common::Force<Ent, Ax2>& F2)
37 {
38    return m_products.setData("F2", F2);
39 }
```

Enfin, l'Exemple de Code D.11 présente un exemple de codage de *Physique*.

Exemple de Code D.11 – Physique de la classe ResultingForceComputation

```
1  template <common::frame Ent, common::axis Ax1, common::axis Ax2>
2  common::errorCode ResultingForceComputation<Ent, Ax1, Ax2>::physics()
3  {
4     common::errorCode err; // Empty error list
5     common::Force<Ent, Ax1> F1;
6     common::Force<Ent, Ax2> F2;
7
8     common::force f1 = 0;
9     common::force f2 = 0;
10
11    for(unsigned int i = 0; i < get_need_Npts(&err).toNoUnit(); ++i)
12    {
13       f1 -= get_need_KDVZ(&err).toNewtonperMeter() *
              get_need_intru(&err).get_point(i).r().toMeter() *
              cos(get_need_intru(&err).get_point(i).theta().toRadian());
14       f2 -= get_need_KDVZ(&err).toNewtonperMeter() *
              get_need_intru(&err).get_point(i).r().toMeter() *
              sin(get_need_intru(&err).get_point(i).theta().toRadian());
15    }
16
17    err += F1.fromNewton(f1);
18    err += F2.fromNewton(f2);
19
20    // Update Products value
21    err += set_product_F1(F1);
22    err += set_product_F2(F2);
23
24    return err; // Return list of execution errors
```

```
25 }
```

Comme on peut le voir sur cet exemple, les erreurs d'exécution sont récoltées au fur et à mesure. Les *Besoins* sont récupérés via les accesseurs dédiés et, en fin de méthode, la valeur des *Produits* est mise à jour via les accesseurs dédiés.

Exemple D.2:

Voyons maintenant un exemple d'implémentation de *Molécule simple*. Considérons celle présentée dans l'Exemple B.4, chargée d'effectuer un changement de repère en coordonnées cylindriques. Sa définition est donnée dans l'Exemple de Code D.12.

Exemple de Code D.12 – Définition de la classe CylindricalFrameShift

```
1  #include "CylindricalToCartesian.h"
2  #include "FrameShiftCartesian.h"
3  #include "CartesianToCylindrical.h"
4
5  namespace Molecule
6  {
7     template <common::frame fr_orig, common::frame fr_dest>
8     class CylindricalFrameShift : public common::abstractEntity
9     {
10    public :
11       CylindricalFrameShift();
12       virtual ~CylindricalFrameShift();
13       virtual common::errorCode physics();
14
15       common::errorCode set_need_Npts(const common::UInteger<common::NOFRAME,
             common::NOAXIS>& Npts);
16       common::errorCode set_need_Cylin(const common::Array<
             common::CylindricPoint<fr_orig> >& Cylin);
17       common::errorCode set_need_frame(const common::CartesianPose<fr_dest>& frame);
18       common::errorCode set_need_or_y(const common::Angle<fr_dest, common::y>& or_y);
19       common::errorCode set_need_or_z(const common::Angle<fr_dest, common::z>& or_z);
20
21       common::Array< common::CylindricPoint<fr_dest> >
             get_product_Cylout(common::errorCode* e = NULL) const;
22
23    protected :
```

```
24      Utilities::MathUtils::CylindricalToCartesian<fr_orig> CylToCart;
25      Utilities::MathUtils::FrameShiftCartesian<fr_orig, fr_dest> FS;
26      Utilities::MathUtils::CartesianToCylindrical<fr_dest> CartToCyl;
27    };
28 }
29
30 #include "CylindricalFrameShift.aid"
```

Nous avons vu que les *Molécules* ne sont pas rattachées à un *Domaine de Connaissance* puisqu'elles sont susceptibles d'incorporer des *Entités Composables* provenant de domaines très variés. Néanmoins, nous les associons à un *namespace* dédié (ligne 5) qui permet d'indiquer explicitement à un utilisateur qu'il s'agit d'une *Molécule*.

Les entités utilisées dans la *Molécule* doivent être déclarées comme variables internes à celle-ci (ligne 24 à 26).

La gestion des *Eléments d'Interface* est par contre différente de celle des *Atomes*. En effet, comme indiqué dans la Définition 34, un *Lien d'Interface* est un lien de référencement. De fait seules les méthodes permettant leur accès depuis l'extérieur de la classe (écriture des *Besoins*, lecture des *Produits* et lecture et écriture des *Stockages Internes*) sont utilisées. En outre, nous ne créons plus d'*Eléments d'Interface* pour les *Molécules* car les différentes méthodes d'accès implémenteront directement le lien de référencement, ce qui permet de gagner en espace mémoire occupé et en temps de calcul en évitant des copies de données qui ne sont pas nécessaires.

Dès lors, le constructeur de la classe est désormais vide, tout comme son destructeur (Exemple de Code D.13).

Exemple de Code D.13 – Constructeur et Destructeur de la classe CylindricalFrameShift

```
1 template <common::frame fr_orig, common::frame fr_dest>
2 CylindricalFrameShift<fr_orig, fr_dest>::CylindricalFrameShift() :
      common::abstractEntity()
3 {
4 }
5
6 template <common::frame fr_orig, common::frame fr_dest>
7 CylindricalFrameShift<fr_orig, fr_dest>::~CylindricalFrameShift()
8 {
9 }
```

En outre, les méthodes accesseurs font directement le lien de référencement avec l'*Interface* des entités contenues dans la *Molécule*, comme l'illustre l'Exemple de Code D.14, tout en permettant de transmettre les erreurs ayant pu se produire lors de la mise à jour des données.

Exemple de Code D.14 – Accesseurs de la classe CylindricalFrameShift

```
1  template <common::frame fr_orig, common::frame fr_dest>
2  common::errorCode CylindricalFrameShift<fr_orig, fr_dest>::set_need_Npts(const
       common::UInteger<common::NOFRAME, common::NOAXIS>& Npts)
3  {
4     common::errorCode err;
5     err += CylToCart.set_need_Npts(Npts);
6     err += FS.set_need_Npts(Npts);
7     err += CartToCyl.set_need_Npts(Npts);
8     return err;
9  }
10
11 template <common::frame fr_orig, common::frame fr_dest>
12 common::errorCode CylindricalFrameShift<fr_orig, fr_dest>::set_need_Cylin(const
       common::Array< common::CylindricPoint<fr_orig> >& Cylin)
13 {
14    common::errorCode err;
15    err += CylToCart.set_need_in(Cylin);
16    return err;
17 }
18
19 template <common::frame fr_orig, common::frame fr_dest>
20 common::errorCode CylindricalFrameShift<fr_orig, fr_dest>::set_need_frame(const
       common::CartesianPose<fr_dest>& frame)
21 {
22    common::errorCode err;
23    err += FS.set_need_frame(frame);
24    return err;
25 }
26
27 template <common::frame fr_orig, common::frame fr_dest>
28 common::errorCode CylindricalFrameShift<fr_orig, fr_dest>::set_need_or_y(const
       common::Angle<fr_dest, common::y>& or_y)
29 {
```

```
30    common::errorCode err;
31    err += CartToCyl.set_need_or_y(or_y);
32    return err;
33 }
34
35 template <common::frame fr_orig, common::frame fr_dest>
36 common::errorCode CylindricalFrameShift<fr_orig, fr_dest>::set_need_or_z(const
       common::Angle<fr_dest, common::z>& or_z)
37 {
38    common::errorCode err;
39    err += CartToCyl.set_need_or_z(or_z);
40    return err;
41 }
42
43 template <common::frame fr_orig, common::frame fr_dest>
44 common::Array< common::CylindricPoint<fr_dest> > CylindricalFrameShift<fr_orig,
       fr_dest>::get_product_Cylout(common::errorCode* e) const
45 {
46    return CartToCyl.get_product_out(e);
47 }
```

Enfin la *Physique* de la *Molécule simple* fait appel aux *Physiques* des différentes entités qu'elle contient et assure également la réalisation des *Liens* entre entités comme le montre l'Exemple de Code D.15. Cet exemple nous permet également de montrer comment peut être fait un *Lien* entre deux entités logicielles avec notre API (lignes 7 et 10).

Exemple de Code D.15 – Physique de la classe CylindricalFrameShift

```
1  template <common::frame fr_orig, common::frame fr_dest>
2  common::errorCode CylindricalFrameShift<fr_orig, fr_dest>::physics()
3  {
4     common::errorCode err;
5     /// Entity Physics ///
6     err += CylToCart.physics();
7     err += FS.set_need_CartIn(CylToCart.get_product_out(&err)); // Link
8
9     err += FS.physics();
10    err += CartToCyl.set_need_in(FS.get_product_CartOut(&err)); // Link
11
```

```
12    err += CartToCyl.physics();
13
14    return err;
15 }
```

Comme illustré par ces deux exemples, la structure de nos différentes entités logicielles est très normalisée et le code à écrire est donc chaque fois d'une structure identique.

D.2 Description générique de nos entités

Nous avons décrit l'API utilisée pour mettre en œuvre nos *Entités Composables*. Néanmoins, il ne s'agit que d'une implémentation possible, qui ne peut servir de référence dans la description de nos entités. Il nous faut donc également disposer d'un moyen de décrire nos entités avec un niveau d'abstraction suffisant pour que celles-ci soient indépendantes de l'implémentation qui en sera ensuite faite. Cette description doit faciliter les échanges d'*Entités Composables* entre les différentes parties prenantes lors de la conception du contrôleur du robot et simplifier également leur diffusion. Enfin, cette description, couplée à un outil de génération de code, permettrait d'accélérer l'implémentation des entités logicielles sur les différentes cibles applicatives.

D.2.1 Description proposée

Le premier point à traiter est le choix du langage pour notre description. Deux options se dégagent.

Premièrement, il est possible de se baser sur un langage existant tel que l'*Extensible Markup Language* (XML) [BPSM+08]. Celui-ci permet de spécifier une syntaxe spécifique pour les balises utilisées dans la description, soit via un schéma XML (XSD) soit via une Définition de Type de Document (DTD). Ces définitions permettent une modification aisée de la sémantique de notre description, et des bibliothèques telles que Xerces [2] offrent des fonctionnalités de *parsing* [3] et de vérification syntaxique. En plus de ces bibliothèques, certains Environnements de Développement Intégrés (IDE) tels qu'Eclipse [4] comprennent des plugins

2. http ://xerces.apache.org/xerces-c/

3. Le *parsing* est le fait d'analyser un fichier grammaticalement structuré pour récupérer les informations qu'il contient.

4. https ://www.eclipse.org/ide/

dédiés XML permettant d'assister l'utilisateur dans l'écriture de fichiers et proposant une vérification syntaxique en ligne lors de l'édition du fichier. De fait, toute évolution syntaxique apportée à notre description est facilitée. Un autre avantage est sa structure et sa syntaxe facilement compréhensibles par un humain même si ce dernier n'est pas habitué à la programmation. En revanche, l'utilisation des balises a tendance à donner une syntaxe plus lourde et souvent répétitive à écrire.

La seconde option est le développement d'un langage dédié. Celui-ci, conçu spécifiquement pour nos besoins, peut s'avérer plus intuitif à utiliser et d'une syntaxe plus légère. Néanmoins, il nécessite le développement d'un *parser* dédié. Ce dernier peut s'appuyer sur des outils d'analyse lexicale et syntaxique tels que Flex et Bison [Aab03]. Néanmoins cette solution est aussi moins évolutive puisque toute modification syntaxique ou sémantique nécessite d'adapter l'analyseur en conséquence.

Enfin, si l'*Interface* peut être décrite avec ces outils, le cas de la *Physique* de nos *Entités Composables* est plus délicat. Une description générique peut être intéressante à la fois pour réduire le risque d'erreur lors de la production des entités logicielles mais aussi parce qu'elle offre des perspectives intéressantes pour, par exemple, estimer le temps de calcul de nos entités (en effet, connaissant les opérations élémentaires à réaliser et leur nombre et sachant le coût temporel de chacune d'entre elles sur notre cible, il est envisageable d'avoir une estimation du temps de calcul utilisé). Mais pour l'obtenir, il faut pouvoir proposer en ensemble d'opérateurs qui permettent de réaliser les calculs souhaités. Dans les deux cas, il s'agit donc d'une représentation dédiée en plus de celle utilisée pour décrire la structure de nos entités. En outre, dans le cas d'une description XML, il est très difficile de décrire la *Physique* avec des balises et dès lors sa description devrait être traitée séparément de celle du reste de l'entité.

Le Tableau D.1 résume les avantages et inconvénients de chaque méthode.

Dans notre cas, étant donné que l'une des principales tâches effectuées dans nos travaux a été de proposer un formalisme et la sémantique associée, la syntaxe de notre représentation générique a été amenée à évoluer. Dans ce contexte, où la souplesse de modification était l'élément primordial, nous avons choisi de proposer une description générique de nos entités en XML.

Nous définissons pour cela notre syntaxe à l'aide de différents schémas XML qui permettent de définir la syntaxe de notre description. Cette syntaxe se focalise sur l'*Interface* de nos entités. La *Physique* ne sera, quant à elle, pas décrite et son codage sera à la charge de l'utilisateur.

Nous proposons ci-après des exemples illustrant cette description générique pour les *Atomes*,

Tableau D.1 – Avantages et inconvénients des différentes options

Option	Avantages	Inconvénients
Adaptation d'un langage existant (XML)	*Parsing* simple grâce aux outils existants Evolutivité facilitée par les outils génériques si la sémantique ou la syntaxe évoluent Compréhension et écriture possible même sans habitude de la programmation	Syntaxe lourde Obligation d'utiliser un langage supplémentaire pour la *Physique*
Langage dédié	Syntaxe plus légère et intuitive Possibilité d'intégrer la description de la *Physique*	Nécessite le développement d'un *parser* dédié ce qui pénalise l'évolutivité

Molécules et *Alternatives*. Il est, par contre, important de noter que nous n'avons, pour l'instant, pas défini de description générique pour les *Compositions*.

D.2.2 Description générique d'un Atome

Pour illustrer la description générique d'un *Atome*, nous allons nous baser sur l'entité utilisée dans les Exemples A.2 et D.1. Sa description générique, sous forme de fichier XML, est donnée dans l'Exemple de Code D.16.

Exemple de Code D.16 – Exemple de description générique d'un Atome

```
1 <?xml version="1.0" encoding="UTF-8"?>
2 <tns:Atom xmlns:tns="atom.namespace"
      xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
      xsi:schemaLocation="atom.namespace Atom.xsd">
3    <tns:Name>ResultingForceComputation</tns:Name>
4    <tns:InterfaceParameter name="Ent" type="frame"/>
5    <tns:InterfaceParameter name="Ax1" type="axis"/>
6    <tns:InterfaceParameter name="Ax2" type="axis"/>
7    <tns:Interface>
```

```
8       <tns:Need>
9          <tns:Name>Npts</tns:Name>
10         <tns:Datatype name="UInteger"/>
11         <tns:Frame>
12            <tns:FValue>NOFRAME</tns:FValue>
13         </tns:Frame>
14         <tns:Axis>
15            <tns:AValue>NOAXIS</tns:AValue>
16         </tns:Axis>
17      </tns:Need>
18      <tns:Need>
19         <tns:Name>KDVZ</tns:Name>
20         <tns:Datatype name="Stiffness"/>
21         <tns:Frame>
22            <tns:FValue>NOFRAME</tns:FValue>
23         </tns:Frame>
24         <tns:Axis>
25            <tns:AValue>NOAXIS</tns:AValue>
26         </tns:Axis>
27      </tns:Need>
28      <tns:Need>
29         <tns:Name>intru</tns:Name>
30         <tns:Datatype name="CylindricPoint" array="true"/>
31         <tns:Frame>
32            <tns:Parameter>Ent</tns:Parameter>
33         </tns:Frame>
34      </tns:Need>
35      <tns:Product>
36         <tns:Name>F1</tns:Name>
37         <tns:Datatype name="Force"/>
38         <tns:Frame>
39            <tns:Parameter>Ent</tns:Parameter>
40         </tns:Frame>
41         <tns:Axis>
42            <tns:Parameter>Ax1</tns:Parameter>
43         </tns:Axis>
44      </tns:Product>
```

```
45      <tns:Product>
46         <tns:Name>F2</tns:Name>
47         <tns:Datatype name="Force"/>
48         <tns:Frame>
49            <tns:Parameter>Ent</tns:Parameter>
50         </tns:Frame>
51         <tns:Axis>
52            <tns:Parameter>Ax2</tns:Parameter>
53         </tns:Axis>
54      </tns:Product>
55   </tns:Interface>
56   <tns:KnowledgeDomain>
57      <tns:Environment>
58         <tns:Karst/>
59      </tns:Environment>
60   </tns:KnowledgeDomain>
61 </tns:Atom>
```

L'Exemple de Code D.16 présente la description de l'*Atome ResultingForceComputation*. Les balises aux lignes 2 et 61 servent à délimiter la définition de notre entité qui se conforme à la description proposée par le schéma *Atom.xsd*. Le nom de l'entité est défini ligne 3. Les différents *Paramètres d'Interface* sont ensuite définis. L'attribut *name* sert à indiquer leur nom et l'attribut *type* indique s'il s'agit d'un paramètre portant sur un axe ou un repère.

Chaque élément de l'*Interface* est ensuite défini à l'aide de 4 balises. La balise $Name$ définit son nom, *Datatype* définit le *Type* via l'attribut *name* et un attribut optionnel *array* peut être mis à *true* pour indiquer qu'il s'agit d'un *TypeArray*. En l'absence de cet attribut ou s'il est mis à $false$, il s'agit d'un *Type* normal. Repère et axe sont définis via des balises spécifiques qui contiennent soit une balise $Value$ soit une balise $Parameter$. La seconde sert à indiquer qu'un des *Paramètres d'Interface* est utilisé. Les balises $FValue$ et $AValue$ permettent de valuer respectivement repère et axes à partir d'une énumération préalablement définie. Enfin, le *Domaine de Connaissance* est défini par une succession de balises.

Enfin, nous devons préciser que les *Connaissances Externes* sont décrites de la même manière que les *Atomes* (Exemple de Code D.16). Seule la ligne 2 change puisque la balise *Atom* est remplacée par la balise $ExtKnowledge$.

D.2.3 Description générique d'une Molécule

Pour illustrer la description générique d'une *Molécule*, nous allons nous baser sur l'entité utilisée dans les Exemples B.4 et D.2. Sa description générique, sous forme de fichier XML, est donnée dans l'Exemple de Code D.17.

Exemple de Code D.17 – Exemple de description générique d'une Molécule

```
<?xml version="1.0" encoding="UTF-8"?>
<tns:Molecule xmlns:tns="atom.namespace"
    xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
    xsi:schemaLocation="atom.namespace Molecule.xsd">
  <tns:Name>CylindricalFrameShift</tns:Name>
  <tns:InterfaceParameter name="fr_orig" type="frame"/>
  <tns:InterfaceParameter name="fr_dest" type="frame"/>
  <tns:Interface>
    <tns:Need>
      <tns:Name>Npts</tns:Name>
      <tns:Datatype name="UInteger"/>
      <tns:Frame>
        <tns:FValue>NOFRAME</tns:FValue>
      </tns:Frame>
      <tns:Axis>
        <tns:AValue>NOAXIS</tns:AValue>
      </tns:Axis>
    </tns:Need>
    <tns:Need>
      <tns:Name>Cylin</tns:Name>
      <tns:Datatype name="CylindricPoint" array="true"/>
      <tns:Frame>
        <tns:Parameter>fr_orig</tns:Parameter>
      </tns:Frame>
    </tns:Need>
    <tns:Need>
      <tns:Name>frame</tns:Name>
      <tns:Datatype name="CartesianPose"/>
      <tns:Frame>
        <tns:Parameter>fr_dest</tns:Parameter>
      </tns:Frame>
```

```
30      </tns:Need>
31      <tns:Need>
32         <tns:Name>or_y</tns:Name>
33         <tns:Datatype name="Angle"/>
34         <tns:Frame>
35            <tns:Parameter>fr_dest</tns:Parameter>
36         </tns:Frame>
37         <tns:Axis>
38            <tns:AValue>y</tns:AValue>
39         </tns:Axis>
40      </tns:Need>
41      <tns:Need>
42         <tns:Name>or_z</tns:Name>
43         <tns:Datatype name="Angle"/>
44         <tns:Frame>
45            <tns:Parameter>fr_dest</tns:Parameter>
46         </tns:Frame>
47         <tns:Axis>
48            <tns:AValue>z</tns:AValue>
49         </tns:Axis>
50      </tns:Need>
51      <tns:Product>
52         <tns:Name>Cylout</tns:Name>
53         <tns:Datatype name="CylindricPoint" array="true"/>
54         <tns:Frame>
55            <tns:Parameter>fr_dest</tns:Parameter>
56         </tns:Frame>
57      </tns:Product>
58   </tns:Interface>
59   <tns:Entities>
60      <tns:Entity type="CartesianToCylindrical" name="CartToCyl">
61         <tns:SetParam name="Ent">
62            <tns:Parameter>fr_dest</tns:Parameter>
63         </tns:SetParam>
64      </tns:Entity>
65      <tns:Entity type="FrameShiftCartesian" name="FS">
66         <tns:SetParam name="fr_orig">
```

```
67            <tns:Parameter>fr_orig</tns:Parameter>
68         </tns:SetParam>
69         <tns:SetParam name="fr_dest">
70            <tns:Parameter>fr_dest</tns:Parameter>
71         </tns:SetParam>
72      </tns:Entity>
73      <tns:Entity type="CylindricalToCartesian" name="CylToCart">
74         <tns:SetParam name="Ent">
75            <tns:Parameter>fr_orig</tns:Parameter>
76         </tns:SetParam>
77      </tns:Entity>
78   </tns:Entities>
79   <tns:EntityLinks>
80      <tns:Link source="CylToCart" sProduct="out" dest="FS" dNeed="CartIn" />
81      <tns:Link source="FS" sProduct="CartOut" dest="CartToCyl" dNeed="in" />
82   </tns:EntityLinks>
83   <tns:InterfaceLinks>
84      <tns:InLink need="Npts" dest="CylToCart" dNeed="Npts"/>
85      <tns:InLink need="Npts" dest="FS" dNeed="Npts"/>
86      <tns:InLink need="Npts" dest="CartToCyl" dNeed="Npts"/>
87      <tns:InLink need="or_y" dest="CartToCyl" dNeed="or_y"/>
88      <tns:InLink need="or_z" dest="CartToCyl" dNeed="or_z"/>
89      <tns:InLink need="Cylin" dest="CylToCart" dNeed="in"/>
90      <tns:InLink need="frame" dest="FS" dNeed="frame"/>
91      <tns:OutLink product="Cylout" source="CartToCyl" sProduct="out"/>
92   </tns:InterfaceLinks>
93 </tns:Molecule>
```

L'Exemple de Code D.17 présente la description de la *Molécule* $CylindricalFrameShift$. La balise contenante (ligne 2) vient cette fois-ci utiliser le schéma $Molecule.xsd$ qui définit la description d'une *Molécule*. La définition du nom, des *Paramètres d'Interface* et de l'*Interface* est identique à celle d'un *Atome*. Ensuite, vient la définition des éléments contenus dans la *Physique*.

Les différentes *Entités Composables* contenues dans la *Physique* sont définies entre les balises *Entities* (lignes 59 et 78) à l'aide d'une balise *Entity*. Celle-ci contient deux attributs : *type* qui sert à préciser le nom de l'*Entité Composable* utilisée, et *name* qui représente son identifiant. La balise $SetParam$ permet de fixer la valeur du *Paramètre d'Interface* indiqué par

l'attribut *name* de l'entité. Celui-ci peut soit être directement valué via une des balises $FValue$ ou $AValue$ précédemment définie soit lié à un des *Paramètres d'Interface* de la *Molécule* via la balise *Parameter*.

La balise *EntityLinks* délimite la définition des *Liens* entre les entités (lignes 79 à 82). Ceux-ci sont définis via une balise *Link* qui utilise 4 attributs. Le premier, *source*, doit contenir l'identifiant de l'entité dont provient le lien et *dest* celui de l'entité destinataire. *sProduct* et $dNeed$ désignent respectivement le *Produit* et le *Besoin* qui sont connectés ensemble par le *Lien*. Evidemment, *sProduct* doit être un *Produit* de *source* et $dNeed$ un *Besoin* de *dest*.

Les balises $InterfaceLinks$ sont utilisées pour définir les *Liens d'Interface* (ligne 83 à 92). Dans notre description de l'entité, afin de simplifier son analyse et sa vérification syntaxique, nous avons séparé les *Liens d'Interface* reliant un *Besoin* de la *Molécule* (attribut *need*) et le *Besoin* (attribut $dNeed$) d'une entité qu'elle contient (attribut *dest*) représentés par la balise *InLink* de ceux reliant un *Produit* de la *Molécule* (attribut *product*) et le *Produit* (attribut *sProduct*) d'une entité qu'elle contient (attribut *source*) représentés par la balise *OutLink*.

D.2.4 Description générique d'une Alternative

L'Exemple de Code D.18 présente la description de l'*Alternative AngleManagement* présentée dans l'Exemple B.8 afin d'illustrer la description des *Alternatives*.

Exemple de Code D.18 – Exemple de description générique d'une Alternative

```
1  <?xml version="1.0" encoding="UTF-8"?>
2  <tns:Alternative xmlns:tns="atom.namespace"
       xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
       xsi:schemaLocation="atom.namespace Alternative.xsd">
3     <tns:Name>AngleManagement</tns:Name>
4     <tns:InterfaceParameter name="Ent" type="frame"/>
5     <tns:InterfaceParameter name="ax" type="axis"/>
6     <tns:Interface>
7        <tns:Need>
8           <tns:Name>angle_meas</tns:Name>
9           <tns:Datatype name="Angle"/>
10          <tns:Frame>
11             <tns:Parameter>Ent</tns:Parameter>
12          </tns:Frame>
13          <tns:Axis>
14             <tns:Parameter>ax</tns:Parameter>
```

```
15          </tns:Axis>
16       </tns:Need>
17       <tns:Need>
18          <tns:Name>t_meas</tns:Name>
19          <tns:Datatype name="Duration"/>
20          <tns:Frame>
21             <tns:FValue>NOFRAME</tns:FValue>
22          </tns:Frame>
23          <tns:Axis>
24             <tns:AValue>NOAXIS</tns:AValue>
25          </tns:Axis>
26       </tns:Need>
27          <tns:Need>
28          <tns:Name>spe_meas</tns:Name>
29          <tns:Datatype name="AngularSpeed"/>
30          <tns:Frame>
31             <tns:Parameter>Ent</tns:Parameter>
32          </tns:Frame>
33          <tns:Axis>
34             <tns:Parameter>ax</tns:Parameter>
35          </tns:Axis>
36       </tns:Need>
37       <tns:Need>
38          <tns:Name>useControl</tns:Name>
39          <tns:Datatype name="Boolean"/>
40          <tns:Frame>
41             <tns:FValue>NOFRAME</tns:FValue>
42          </tns:Frame>
43          <tns:Axis>
44             <tns:AValue>NOAXIS</tns:AValue>
45          </tns:Axis>
46       </tns:Need>
47       <tns:Product>
48          <tns:Name>accel</tns:Name>
49          <tns:Datatype name="AngularAcceleration"/>
50          <tns:Frame>
51             <tns:Parameter>Ent</tns:Parameter>
```

```
52          </tns:Frame>
53          <tns:Axis>
54             <tns:Parameter>ax</tns:Parameter>
55          </tns:Axis>
56       </tns:Product>
57    </tns:Interface>
58    <tns:Entities>
59       <tns:Entity type="TeleopAngularAccel" name="e1">
60          <tns:SetParam name="Ent">
61             <tns:Parameter>Ent</tns:Parameter>
62          </tns:SetParam>
63          <tns:SetParam name="ax">
64             <tns:Parameter>ax</tns:Parameter>
65          </tns:SetParam>
66       </tns:Entity>
67       <tns:Entity type="AngleControl" name="e2">
68          <tns:SetParam name="Ent">
69             <tns:Parameter>Ent</tns:Parameter>
70          </tns:SetParam>
71          <tns:SetParam name="ax">
72             <tns:Parameter>ax</tns:Parameter>
73          </tns:SetParam>
74       </tns:Entity>
75    </tns:Entities>
76    <tns:SelectionEntity>
77       <tns:Entity type="SelectorAngleManagement" name="e_sel"/>
78    </tns:SelectionEntity>
79    <tns:JunctionEntities>
80       <tns:JunctionEntity type="InitAngleControl" name="e12" from="e1" to="e2">
81          <tns:SetParam name="Ent">
82             <tns:Parameter>Ent</tns:Parameter>
83          </tns:SetParam>
84          <tns:SetParam name="ax">
85             <tns:Parameter>ax</tns:Parameter>
86          </tns:SetParam>
87       </tns:JunctionEntity>
88       <tns:JunctionEntity type="Empty" name="e21" from="e2" to="e1"/>
```

```
89    </tns:JunctionEntities>
90    <tns:SelectionLinks>
91       <tns:Link source="e_sel" sProduct="entityToRun" dest="e12" dNeed="entityToRun"
             />
92    </tns:SelectionLinks>
93    <tns:JunctionLinks>
94       <tns:JunctionLink source="e12" sProduct="angle_ref" dest="e2"
             dStorage="angle_ref" />
95       <tns:JunctionLink source="e12" sProduct="err_pid" dest="e2"
             dStorage="error_tt" />
96    </tns:JunctionLinks>
97    <tns:InterfaceLinks>
98       <tns:InLink need="useControl" dest="e_sel" dNeed="useControl"/>
99       <tns:InLink need="angle_meas" dest="e2" dNeed="angle_meas"/>
100      <tns:InLink need="angle_meas" dest="e12" dNeed="angle_meas"/>
101      <tns:InLink need="spe_meas" dest="e2" dNeed="spe_meas"/>
102      <tns:InLink need="t_meas" dest="e2" dNeed="t_meas"/>
103      <tns:OutLink product="accel" source="e1" sProduct="accel"/>
104      <tns:OutLink product="accel" source="e2" sProduct="accel"/>
105   </tns:InterfaceLinks>
106 </tns:Alternative>
```

La balise contenante (ligne 2) vient cette fois-ci utiliser le schéma *Alternative.xsd*. En supplément des balises $Name$, $InterfaceParameter$ et $Interface$ communes aux autres *Entités Composables*, nous utilisons des balises spécifiques aux *Alternatives*. La balise *Entities* contient la description des entités substituables de l'*Alternative*. La balise *SelectionEntity* contient la description de l'*Entité Composable* de sélection. Les *Entités Composables* de jonction sont définies entre les balises *JunctionEntities* via des balises *JunctionEntity*. Cette dernière balise étend la définition de la balise *Entity* précédemment définie en y ajoutant deux attributs pour indiquer les entités liées par la jonction. Le premier, $from$, sert à indiquer l'entité depuis laquelle la commutation se déroule et *to* indique l'entité active suite à la commutation.

Les *Liens* entre l'entité de sélection et celles de jonction, servant à indiquer l'entité à activer, sont regroupés par les balises *SelectionLinks*. Les balises *JunctionLinks* regroupent les liens entre les *Entités Composables* de jonction et les *Entités Composables* substituables. Ceux-ci sont décrits à l'aide de la balise *JunctionLink* qui comprend quatre attributs. Les deux premiers, *source* et *sProduct* servent à indiquer l'entité source et le *Produit* de celle-ci

tandis que *dest* et *dStorage* servent à indiquer l'entité destinataire et le *Stockage Interne* qui va recevoir le *Lien*. Enfin, les différents *Liens d'Interface* sont décrits de manière identique à ceux d'une *Molécule*.

D.2.5 Utilitaire de génération de code

Comme nous l'avons présenté précédemment, les entités logicielles mises en œuvre à partir de l'API proposée possèdent une structure très normalisée qui se prête bien à une génération automatique. Celle-ci permet un gain de temps significatif dans le développement de nos entités tout en réduisant les risques d'erreurs dûs à l'utilisateur. Nous allons présenter ici l'outil qui, à partir de la description générique de nos entités, permet de les implémenter avec notre API.

Les différentes étapes de la génération automatique de code pour les *Atomes* sont présentées Figure D.3.

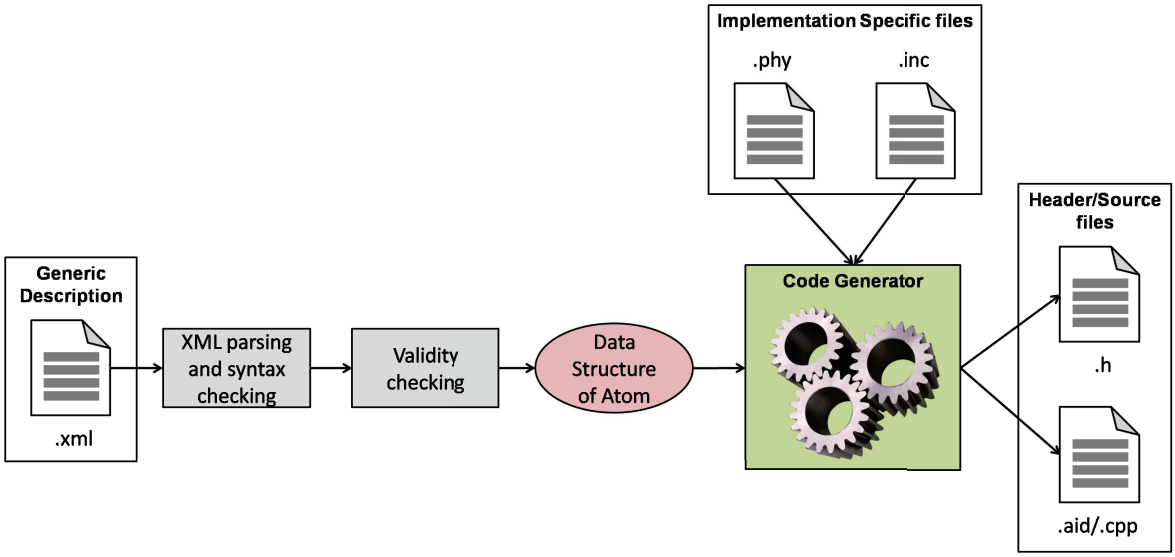

FIGURE D.3 – Vue schématique du déroulement de la génération du code d'un *Atome*

La première étape consiste à extraire les informations du fichier XML. Nous utilisons pour cela la librairie Xerces. Le *parser* permet en outre d'effectuer une vérification syntaxique de notre description à partir du Schéma XML définissant sa syntaxe. Nous réalisons ensuite une étape de validation qui permet de détecter des erreurs qui ne sont pas vérifiables par le *parser* XML car non descriptibles dans le schéma. Cela inclut notamment la vérification que les *Eléments d'Interface* ont des noms différents ou encore que les *Paramètres d'Interface* affectés à nos différents *Eléments d'Interface* correspondent bien à des *Paramètres d'Interface* de notre *Atome*. Si notre description est valide, celle-ci est stockée dans une structure de données qui contient toutes les informations nécessaires à son utilisation.

A partir de ces données, nous pouvons générer le code de l'entité logicielle. Néanmoins, il est nécessaire d'ajouter le code de la *Physique* de l'entité car celui-ci n'est pas, comme nous l'avons expliqué précédemment, donné dans notre description générique. Le générateur de code va donc utiliser deux fichiers : un fichier d'extension *.phy* qui comprend le contenu de la fonction *physics* de nos entités et un second d'extension *.inc* qui contient tous les headers dont a besoin la fonction *physics* pour être utilisée. A partir de cela, nous pouvons générer les fichiers source (*.cpp* si l'entité n'a pas de *Paramètre d'Interface* ou *.aid* si elle en a) et header qui contiennent le code source de notre entité logicielle.

Les différentes étapes de la génération automatique de code pour les *Molécules simples* sont présentées Figure D.4.

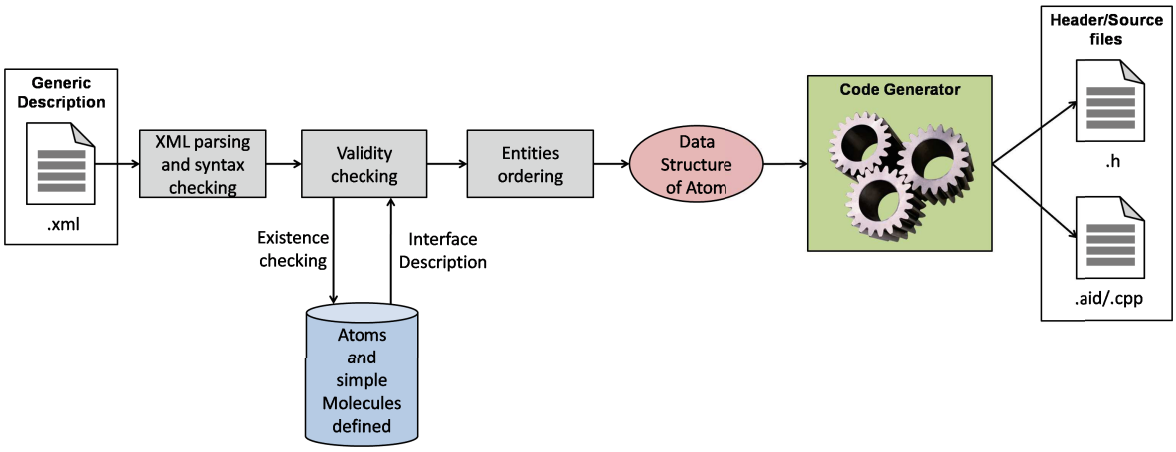

FIGURE D.4 – Vue schématique du déroulement de la génération du code d'une *Molécule simple*

Tout comme pour les *Atomes* nous commençons par extraire les informations du fichier XML. Nous vérifions ensuite la validité de notre entité. Cela nécessite de vérifier que les entités qu'elle utilise existent puis de récupérer la description de leur *Interface*. Cela permet d'appliquer la vérification de validité telle que comprise dans la Définition 36 (cohérence des *Liens*, utilisation des *Besoins* et nombre de *Besoins* et *Produits*).

Il est ensuite nécessaire d'ordonner les entités utilisées dans la *Molécule*, c'est-à-dire de savoir laquelle est la première à s'exécuter puis la seconde et ainsi de suite en se basant sur les connexions entre entités. Pour cela, en supposant que l'intérieur de notre *Molécule* est un graphe orienté, nous utilisons un algorithme de tri topographique. Nous avons choisi l'algorithme basé sur le parcours en profondeur proposé dans [CSRL01]. Néanmoins, comme tous les algorithmes de tri topographique, il impose que le graphe traité soit acyclique afin de pouvoir déterminer une solution au tri d'où la condition d'acyclicité imposée aux *Molécules*

simples. En effet, si le graphe est cyclique, alors il n'est plus possible de savoir quelle est la première entité à devoir s'exécuter.

Nous obtenons ainsi la représentation de notre *Molécule* dans une structure de données qui est ensuite utilisée pour générer le code source de l'entité logicielle l'implémentant.

Il est à noter que notre génération de code présente actuellement une limitation au niveau de la gestion des *Types Array*. En effet, comme présenté dans l'Exemple de Code D.9 lignes 5 à 9, la taille des éléments du tableau doit être fixée par un *Besoin* de l'entité. Nous n'avons, pour l'instant, pas de moyen d'indiquer quel est ce *Besoin* et donc de l'intégrer à la génération de code. Il faudrait donc faire évoluer notre description afin de pouvoir sélectionner ce *Besoin* qui doit obligatoirement être de *Type UInterger*.

Annexe E

Physique de l'Atome VirtualProximeter

Au Chapitre 10, nous avons présenté l'*Atome VirtualProximeter* qui sert à découpler temporellement les *Atomes* implémentant l'évitement de parois de la *Connaissance Externe* associée au sonar profilométrique. Sa *Physique* est donnée dans l'Exemple de Code E.1.

Exemple de Code E.1 – La *Physique* de l'*AtomeVirtualProximeter*

```
bool newMeasSon = false;
for(unsigned int i = 0; i < get_need_Ntr(&err).toNoUnit(); ++i) // Test if the Meas need was updated
{
   if(!(get_need_measSon(&err).get_point(i) == get_storage_measSon_pre(&err).get_point(i))) // If 2 != points
   {
      newMeasSon = true; // New measurement
      err += set_storage_measSon_pre(get_need_measSon(&err)); // Store it
      common::UInteger<common::NOFRAME, common::NOAXIS> cpt_in;
      err += cpt_in.fromNoUnit(std::min(get_storage_cpt_in(&err).toNoUnit() + get_need_Ntr(&err).toNoUnit(), get_need_NR(&err).toNoUnit()));
      err += set_storage_cpt_in(cpt_in); // We increment the number of already received rays
      break;
   }
}

common::Array< common::CylindricPoint<Ent> > meas(get_need_NR(&err).toNoUnit());
if(get_storage_cpt_in(&err).toNoUnit() >= get_need_NR(&err).toNoUnit()) // Simple test assuming we can't receive two identical rays before receiving all rays
{
```

```
18  std::vector<unsigned int> i_in;
19  if(newMeasSon) // If new input, build list of modified rays
20  {
21      for(unsigned int i=0; i < get_need_Ntr(&err).toNoUnit(); ++i)
22      {
23      common::Angle<Ent, Ax> an;
24      err += an.fromRadian(get_need_measSon(&err).get_point(i).theta().toRadian());
25      i_in.push_back(get_need_angles(&err).find(an));
26      }
27  }
28
29  common::duration dt = get_need_t(&err).toSecond() -
        get_storage_t_pre(&err).toSecond(); // No need to check for t_pre update
        since there are at least two rays so we can't use it before it is updated
30
31  for(unsigned int i = 0; i < get_need_NR(&err).toNoUnit(); ++i)
32  {
33      std::vector<unsigned int>::iterator it = std::find(i_in.begin(), i_in.end(),
            i); // Find if ray has been updated
34      if(it != i_in.end()) //If updated, copy directly input
35      {
36      unsigned int d = std::distance(i_in.begin(), it);
37      meas.set_point(get_need_measSon(&err).get_point(d), i);
38      }
39      else // Otherwise update from robot motion
40      {
41      common::angle alpha =
            get_storage_meas_pre(&err).get_point(i).theta().toRadian();
42      common::distance d1 = - get_need_speed1(&err).toMeterperSecond() * dt; //
            Speed integration
43      common::distance d2 = - get_need_speed2(&err).toMeterperSecond() * dt;
44
45      common::distance d = get_storage_meas_pre(&err).get_point(i).r().toMeter() +
            d1 * cos(alpha) + d2 * sin(alpha); // COmpute new distance
46
47      common::CylindricPoint<Ent> p;
48      err += p.r().fromMeter(d);
```

```
49          err += p.theta().fromRadian(alpha);
50          err += p.h().fromMeter(get_storage_meas_pre().get_point(i).h().toMeter());
51          err +=
                p.or_y().fromRadian(get_storage_meas_pre().get_point(i).or_y().toRadian());
52          err +=
                p.or_z().fromRadian(get_storage_meas_pre().get_point(i).or_z().toRadian());
53
54          err += meas.set_point(p, i);
55          }
56      }
57      err += set_storage_meas_pre(meas);
58  }
59  else // if the sonar full range is not complete
60  {
61      // Output is set to 0 while we don't have received all the expected sonar rays
62      for(unsigned int i = 0; i < get_need_NR(&err).toNoUnit(); ++i)
63      {
64          common::CylindricPoint<Ent> p;
65          p.r().fromMeter(0.0);
66          p.theta().fromRadian(0.0);
67          p.h().fromMeter(0.0);
68          p.or_y().fromRadian(0.0);
69          p.or_z().fromRadian(0.0);
70
71          meas.set_point(p, i);
72      }
73
74      if(newMeasSon)
75      {
76          //We store the newly received inputs
77          common::Array< common::CylindricPoint<Ent> > meas_pre =
                get_storage_meas_pre(&err);
78          for(unsigned int i = 0; i < get_need_Ntr(&err).toNoUnit(); ++i)
79          {
80          common::Angle<Ent, Ax> an;
81          err += an.fromRadian(get_need_measSon(&err).get_point(i).theta().toRadian());
                // Angle of the ray
```

```
82        int pos = get_need_angles(&err).find(an); // Find its position in the rays
              array
83        meas_pre.set_point(get_need_measSon(&err).get_point(i), pos);
84        }
85        err += set_storage_meas_pre(meas_pre);
86    }
87 }
88
89 err += set_storage_t_pre(get_need_t(&err));
90 err += set_product_meas(meas);
```

La première partie (lignes 1 à 13) sert à vérifier si l'entrée provenant du sonar a changé. Si c'est le cas, la nouvelle mesure est stockée pour comparaison au cycle suivant et la variable $newMeasSon$ est mise à $true$. En plus, un compteur du nombre de rayons reçus est incrémenté.

Nous vérifions ensuite si ce compteur a atteint le nombre de rayons du sonar ce qui signifie que le scan est complet et peut être utilisé pour l'évitement de parois. Si ce n'est pas le cas, nous commençons par fixer la distance de tous les rayons à 0 (lignes 62 à 72) afin d'indiquer que la mesure est invalide. Comme nous l'avons expliqué Chapitre 10, cela nous permet de désactiver la fonctionnalité d'évitement de parois et de rendre le contrôle du ou des degrés de liberté impliqués à l'opérateur (choix effectué au Chapitre 10) ou à permettre l'activation d'une autre fonctionnalité suivant la conception de l'application. Ensuite si la mesure est nouvelle, elle est stockée dans la mesure actuelle (lignes 74 à 87).

Une fois tous les rayons reçus, nous commençons par construire la liste des rayons modifiés (lignes 19 à 27). Pour chaque impact, nous regardons ensuite s'il a été modifié. Si c'est le cas, la nouvelle mesure devient la distance d'impact actuelle.

Sinon nous utilisons le déplacement du robot pour estimer la nouvelle distance mesurée sur le rayon (lignes 41 à 54). En l'absence d'un modèle plus complexe, nous supposons en outre que l'environnement reste constant entre deux mesures effectuées sur un rayon donné. Dès lors la variation de la distance d'impact ne dépend que du déplacement du robot.

Pour cela nous calculons le déplacement effectué depuis la dernière exécution de l'*Atome* à partir de la vitesse du robot (lignes 42 et 43) puis calculons la nouvelle distance (ligne 45).

Publications et Présentations de l'auteur

Adrien Lasbouygues, Benoit Ropars, Robin Passama, David Andreu, and Lionel Lapierre. **Atoms based control of mobile robots with Hardware-in-the-Loop validation**. In *Intelligent Robots and Systems (IROS 2015), IEEE International Conference on,*, 2015.

Adrien Lasbouygues, Lionel Lapierre, David Andreu, Josue Lopez Hermoso, Herve Jourde, and Benoit Ropars. **Stable and reactive centering in conduits for karstic exploration**. In *European Control Conference (ECC'14)*, 2014.

Benoit Ropars, Adrien Lasbouygues, Lionel Lapierre, and David Andreu. **Thruster's Dead-zones Compensation for the Actuation System of an Underwater Vehicle**. In *European Control Conference (ECC'15)*, 2015.

Adrien Lasbouygues, Robin Passama, David Andreu, and Lionel Lapierre. **Atoms : from control description to its implementation**. Présentation dans le cadre des *Journées Architectures Logicielles pour la Robotique Autonome, les Systèmes Cyber-Physiques et les Systèmes Auto-Adaptables*, 2014.

Adrien Lasbouygues, Benoit Ropars, David Andreu, and Lionel Lapierre. **Atom Based Control of Mobile Robots for environment exploration**. Présentation dans le cadre des *Journées Nationales de la Robotique Humanoïde et des Architectures de Contrôle en Robotique*, 2014.

Bibliographie

[Aab03] Anthony Aaby. Compiler construction using Flex and Bison, 2003. Disponible à l'adresse : `http://research.microsoft.com/en-us/um/people/rgal/ar_language/external/compiler.pdf`.

[ABC+14] Benedetto Allotta, Fabio Bartolini, Roberto Conti, Riccardo Costanzi, Jonathan Gelli, Niccolò Monni, Marco Natalini, Luca Pugi, and Alessandro Ridolfi. Marta : An auv for underwater cultural heritage, 2014.

[AC14] AC-CESS. *AC-ROV 100 Datasheet*, Février 2014. Disponible à l'adresse : `http://www.ac-cess.com/FileManager/AC-ROV100DatasheetrevH.pdf`.

[ACF+98] Rachid Alami, Raja Chatila, Sara Fleury, Malik Ghallab, and Félix Ingrand. An architecture for autonomy. *The International Journal of Robotics Research*, 17(4) :315–337, 1998.

[ACR+15] Benedetto Allotta, Riccardo Costanzi, Alessandro Ridolfi, Carlo Colombo, Fabio Bellavia, Marco Fanfani, Fabio Pazzaglia, Ovidio Salvetti, Davide Moroni, Maria Antonietta Pascali, et al. The arrows project : adapting and developing robotics technologies for underwater archaeology. In *IFAC workshop on navigation and control of underwater vehicles (NGCUV 2015). Girona, Spain*, 2015.

[ACSW01] Gianluca Antonelli, Stefano Chiaverini, Nilanjan Sarkar, and Michael West. Adaptive control of an autonomous underwater vehicle : experimental results on odin. *Control Systems Technology, IEEE Transactions on*, 9(5) :756–765, 2001.

[AFY08] Gianluca Antonelli, Thor I. Fossen, and Dana R. Yoerger. Underwater robotics. In Bruno Siciliano and Oussama Khatib, editors, *Springer Handbook of Robotics*, pages 987–1008. Springer Berlin Heidelberg, 2008.

[AI93] Mohamed Bensoubaya Adberrahman Iggidr. Stability of discrete-time systems : New criteria and applications to control problems. Technical Report RR-3003, INRIA, 1993.

[Arb15] Arbor. *EmCORE-i2305 Datasheet*, Mars 2015. Disponible à l'adresse : `http://www.arbor.com.tw/datasheet/2014/MB/EmCORE-i2305_DS_20150327.pdf`.

[Bak05] Michel Bakalowicz. Karst groudwater : a challenge for new ressources. *Journal of Hydrogeology*, pages 13 : 148–160, 2005.

[BB01] Andrea Bacciotti and Alessandra Biglio. Some remarks about stability of nonlinear discrete-time control systems. *Nonlinear Differential Equations and Applications NoDEA*, 8(4) :425–438, 2001.

[Ben04] Michael Benjamin. The interval programming model for multi-objective decision making. Technical Report Technical Report AIM-2004-021, Comuter Science and Artificial Intelligence Laboratory, MIT, 2004.

[Ber97] Pascal Bernabé. -174 mètres pour sauver ... un robot ! *Octopus*, 1997. `http://pascalbernabe.com/WordPress/174-metres/`.

[BFNP13] Magnus Bjerkeng, Pietro Falco, Ciro Natale, and Kristin Pettersen. Discrete-time stability analysis of a control architecture for heterogeneous robotic systems. In *Intelligent Robots and Systems (IROS), 2013 IEEE/RSJ International Conference on*, pages 4778–4783, 2013.

[BGL+08] Ananda Basu, Matthieu Gallien, Charles Lesire, Thanh-Hung Nguyen, Saddek Bensalem, Félix Ingrand, and Joseph Sifakis. Incremental component-based construction and verification of a robotic system. In *ECAI*, 2008.

[Bis16] Farnham Bishop. *The Story of the Submarine*. Century, 1916.

[BL12] Marco Bonvini and Alberto Leva. A modelica library for industrial control systems. In *Proceedings of the 9th Modelica Conference*, 2012.

[BOT09] Marcus Baur, Martin Otter, and Bernhard Thiele. Modelica libraries for linear control systems. In *Proceedings of the 7th Modelica Conference*, 2009.

[Bou12] Thierry Bouldoire. Plongée fatale au célèbre Font Estramar, Mai 2012. `http://www.lindependant.fr/2012/05/26/cousteau-et-haroun-tazieff-ont-rendu-celebre-estramar,140724.php`.

[BPSM+08] Tim Bray, Jean Paoli, C. Michael Sperberg-McQueen, Eve Maler, and François Yergeau. *Extensible Markup Language (XML) 1.0 (Fifth Edition), W3C Recommendation*, Novembre 2008. Disponible à l'adresse : `http://www.w3.org/TR/2008/REC-xml-20081126/`.

[Bro88] Roger Ware Brockett. On the computer control of movement. In *Proceedings of IEEE 1988 International Conference on Robotics and Automation*, 1988.

[BS09] Davide Brugali and Patrizia Scandurra. Component-based robotic engineering (part i). *IEEE Robotics & Automation Magazine*, 16(04) :84–96, 2009.

[BS10] Davide Brugali and Azamat Shakhimardanov. Component-based robotic engineering (part ii). *IEEE Robotics & Automation Magazine*, 17(01) :100–112, 2010.

[BSK03] Herman Bruyninckx, Peter Soetens, and Bob Koninckx. The real-time motion control core of the Orocos project. In *Robotics and Automation (ICRA'03), IEEE International Conference on*, 2003.

[BSNL13] Michael Benjamin, Henrik Schmidt, Paul Newman, and John Leonard. Autonomy for unmanned marine vehicles with moos-ivp. In *Marine Robot Autonomy*, pages 47–90. Springer, 2013.

[Car04] Jean-Damien Carbou. *Conception d'une architecture pour la commande à distance d'un robot d'intervention*. PhD thesis, Université Montpellier II, 2004.

[Car14] Gautier Cariou. L'archéologue des profondeurs. *Les Dossiers de la Recherche*, Décembre 2014.

[Cle96] Paul Clements. A survey of architecture description languages. In *Proceedings of the 8th international workshop on software specification and design*, 1996.

[Clé09] Hugo Clément. Le canal du midi se vidange, les « trésors » apparaissent. `http://www.ladepeche.fr/article/2009/11/09/710947-le-canal-du-midi-se-vidange-les-tresors-apparaissent.html`, 2009.

[CLM+04] Carle Côté, Dominic Létourneau, François Michaud, Jean-Marc Valin, Yannick Brosseau, Clément Raïevsky, Mathieu Lemay, and Victor Tran. Code reusability tools for programming mobile robots. In *Intelligent Robots and Systems (IROS), IEEE/RSJ International Conference on*, 2004.

[Col14] Gerald Coley. *BeagleBone Black System Reference Manual - Revision C.1*, Mai 2014.

[COM15] COMEX. *ROV Super Achille*, 2015. Disponible à l'adresse : `http://www.comex.fr/fileadmin/telechargement/MOYENS_ROV_SousMarins_SuperAchille_1PAGE.pdf`.

[Cor15] Intel Corporation. *Intel Atom Processor E3800 Product Family Datasheet*, Janvier 2015. Disponible à l'adresse : `http://www.intel.com/content/www/us/en/embedded/products/bay-trail/atom-e3800-family-datasheet.html`.

[CSH+11] Antonio Ceballos, Lavindra De Silva, Matthieu Herrb, François Felix Ingrand, Anthony Mallet, Alberto Medina, and M. Prieto. Genom as a robotics framework for planetary rover surface operations. *ASTRA*, pages 12–14, 2011.

[CSRL01] Thomas Cormen, Clifford Stein, Ronald Linn Rivest, and Charles Eric Leiserson. *Introduction to Algorithms*. McGraw-Hill Higher Education, 2nd edition, 2001.

[CZSC09] Giuseppe Conte, Silvia Maria Zanoli, David Scaradozzi, and Andrea Caiti. Robotics techniques for data acquisition in underwater archeology. *International Journal of Mechanics and Control (JoMaC)*, 10(1) :45–51, 2009.

[DdQDF04] Warren E. Dixon, Marcio S. de Queiroz, Darren M. Dawson, and T. J. Flynn. Adaptive tracking and regulation of a wheeled mobile robot with controller/update law modularity. *IEEE Transactions on Control Systems Technology*, 12(01) :138–147, 2004.

[dFdV15] Société Spéléologique de Fontaine de Vaucluse. L'épopée du spélénaute (ROV), 2015. `http://www.ssfv.fr/spelenaute/`.

[DJL+08] Nathalie Dörfliger, Hervé Jourde, Bernard Ladouche, Perrine Fleury, Patrick Lachassagne, Yann Conroux, Séverin Pistre, and Arnaud Vestier. Active water management resources of karstic water catchment : the example of Le Lez spring. In *World Water Congress*, 2008.

[DLTT13] Patricia Derler, Edward Lee, Martin Torngren, and Stavros Tripakis. Cyber-physical system design contracts. In *ACM/IEEE 4th International Conference on Cyber-Physical Systems (ICCPS '13)*, 2013.

[DM12] Matthew Dunbabin and Lino Marques. Robots for environmental monitoring : Significant advancements and applications. *Robotics & Automation Magazine, IEEE*, 19(1) :24–39, 2012.

[DP05] Jean-François Deverge and Isabelle Puaut. Safe measurement-based wcet estimation. *WCET'2005*, 2005.

[dQDA99] Marcio S. de Queiroz, Darren M. Dawson, and Manish Argawal. Adaptive control of robot manipulators with controller/update law modularity. *Automatica*, 35 :1379–1390, 1999.

[EB01] Stewart Edgar and Alan Burns. Statistical analysis of wcet for scheduling. In *Real-Time Systems Symposium, 2001. (RTSS 2001). Proceedings. 22nd IEEE*, pages 215–224, 2001.

[EKJ96] Bernard Espiau, Konstantinos Kapellos, and Muriel Jourdan. Formal verification in robotics : Why and how ? In *Robotics Research*, pages 225–236. Springer, 1996.

[EM97] Hilding Elmqvist and Sven Erik Mattsson. Modelica — the next generation modeling language an international design effort. In *Proceedings of the 1st World Congress on Systems Simulation (WCSS)*, 1997.

[EMO98] Hilding Elmqvist, Sven Erik Mattsson, and Martin Otter. Modelica - the new object-oriented modeling language. In *The 12th European Simulation Multiconference (ESM)*, 1998.

[Far12] Jean-Loup Farges. *PROTEUS - Robotic Ontology and Modelling - 3rd version*, Mars 2012. Disponible à l'adresse : `http://www.anr-proteus.fr/sites/default/files/download/Ontology/R1.1.4.3RoboticOntology_and_Modelling_3rdVersion.pdf`.

[FHC97] Sara Fleury, Matthieu Herrb, and Raja Chatila. Genom : A tool for the specification and the implementation of operating modules in a distributed robot architecture. In *Intelligent Robots and Systems (IROS 1997), IEEE International Conference on*, pages 842–848, 1997.

[FLC+09] Perrine Fleury, Bernard Ladouche, Yann Conroux, Hervé Jourde, and Nathalie Dörfliger. Modelling the hydrologic functions of a karst aquifer under active water management – the lez spring. *Journal of Hydrology*, 365(3–4) :235–243, 2009.

[Foo13] Tully Foote. tf : The transform library. In *Technologies for Practical Robot Applications (TePRA), 2013 IEEE International Conference on, Open-Source Software workshop*, Avril 2013.

[FPL05] Jay Farrell, Shuo Pang, and Wei Li. Chemical plume tracing via an autonomous underwater vehicle. *Oceanic Engineering, IEEE Journal of*, 30(2) :428–442, 2005.

[FZL+15] Ridha Fezzani, Benoit Zerr, Michel Legris, Ali Mansour, and Yann Dupas. Swath bathymetric data fusion application to autonomous underwater vehicles. In *OCEANS 2015*, 2015.

[GA02] Frank Grasso and Jelle Atema. Integration of flow and chemical sensing for guidance of autonomous marine robots in turbulent flows. *Environmental Fluid Mechanics*, 2(1-2) :95–114, 2002.

[GDLS15] Nicolas Gobillot, David Doose, Charles Lesire, and Luca Santinelli. Periodic state-machine aware real-time analysis. In *Emerging Technologies and Factory Automation (ETFA'2015)*, 2015.

[GKN15] Emden Ganser, Eleftherios Koutsofios, and Stephen North. *Drawing Graphs with dot*, Janvier 2015. Disponible à l'adresse : `http://www.graphviz.org/pdf/dotguide.pdf`.

[GMG+15] Nicolas Gobillot, Alessandra Melani, Fabrice Guet, Luca Santinelli, Eric Noulard, David Doose, Charles Lesire-Cabaniols, and Jerome Morio. Building system awa-

reness : Cache characterization through probabilities. In *Journées Formalisation des Activités Concurrentes (FAC'2015)*, 2015.

[GPWJ14] Francisco Gutiérrez, Mario Parise, Jo De Waele, and Hervé Jourde. A review on natural and human-induced geohazards and impacts in karst. *Earth-Science Reviews*, 138 :61–88, 2014.

[Gru93] Thomas Robert Gruber. A translation approach to portable ontology specifications. *Knowledge acquisition*, 5(2) :199–220, 1993.

[HCH15] Ben Hixon, Peter Clark, and Hannaneh Hajishirzi. Learning knowledge graphs for question answering through conversational dialog. In *Conference of the North American Chapter of the Association for Computational Linguistics (NAACL), 2015*, 2015.

[Hen00] Thomas Henzinger. The theory of hybrid automata. In Kemal Inan and Robert Kurshan, editors, *Verification of Digital and Hybrid Systems*, volume 170 of *NATO ASI Series*, pages 265–292. Springer Berlin Heidelberg, 2000.

[HHM09] Jeffery Hansen, Scott Hissam, and Gabriel Moreno. Statistical-based wcet estimation and validation. In *Proceedings of the 9th International Workshop on Worst-Case Execution Time (WCET) Analysis*, 2009.

[HJT12] W.P.M.H. Heemels, K.H. Johansson, and P. Tabuada. An introduction to event-triggered and self-triggered control. In *Decision and Control (CDC), 2012 IEEE 51st Annual Conference on*, 2012.

[HM99] João Pedro Hespanha and A. Stephen Morse. Stability of switched systems with average dwell-time. In *Proceedings of the 38th Conference on Decision and Control*, pages 2655–2660, 1999.

[HVKA+00] Dimitris Hristu-Varsakelis, P.S. Krishnaprasad, Sean Andersson, Fumin Zhang, P. Sodre, and L. D'Anna. The MDLe Engine : a software tool for hybrid motion control. Technical Report CDCSS TR 2000-8, Center for Dynamics and Control of Smart Structures, 2000.

[Ifr12] Ifremer. L'epaulard à la découverte des grands fonds, 2012. `http://wwz.ifremer.fr/grands_fonds/Les-moyens/Les-engins/Les-robots/Robots-Ifremer/L-Epaulard`.

[ILLCP07] Félix Ingrand, Simon Lacroix, Solange Lemai-Chenevier, and Frederic Py. Decisional autonomy of planetary rovers. *Journal of Field Robotics*, 24(7) :559–580, 2007.

[Ins10] Woods Hole Oceanographic Institution. *REMUS 600 Specifications*, Janvier 2010. Disponible à l'adresse : `http://www.whoi.edu/main/remus600`.

[JAJ05] Abdellah El Jalaoui, David Andreu, and Bruno Jouvencel. A control architecture for contextual tasks management : application to the AUV taipan. In *Oceans 2005-Europe*, volume 2, pages 752–757, 2005.

[JML+15] Hervé Jourde, Naomi Mazzilli, Nicolas Lecoq, Bruno Arfib, and Dominique Bertin. KARSTMOD : A generic modular reservoir model dedicated to spring discharge modeling and hydrodynamic analysis in karst. In Bartolomé Andreo, Francisco Carrasco, Juan José Durán, Pablo Jiménez, and James LaMoreaux, editors, *Hydrogeological and Environmental Investigations in Karst Systems*, pages 339–344. Springer Berlin Heidelberg, 2015.

[Kar06] Steven Karris. *Introduction to Simulink with engineering applications*. Orchard Publications, 2006.

[KLSV03] Dilsun Kaynar, Nancy Lynch, Roberto Segala, and Frits Vaandrager. Timed i/o automata : A mathematical framework for modeling and analyzing real-time systems. In *In RTSS 2003 : The 24th IEEE International Real-Time Systems Symposium, Cancun,Mexico*, pages 166–177. Society Press, 2003.

[Kon15] Kongsberg. *Hugin Family Brochure*, 2015. Disponible à l'adresse : `http://www.km.kongsberg.com/ks/web/nokbg0397.nsf/AllWeb/A6A2CC361D3B9653C1256D71003E97D5/\$file/HUGIN_Family_brochure_r2_lr.pdf?OpenElement`.

[KRRH09] Wajahat Kazmi, Pere Ridao, David Ribas, and Emili Hernández. Dam wall detection and tracking using a mechanically scanned imaging sonar. In *Robotics and Automation (ICRA'09), 2009. IEEE International Conference on*, pages 3595–3600, 2009.

[KUKW00] Michio Kumagai, Tamaki Ura, Yoji Kuroda, and Ross Walker. New auv designed for lake environment monitoring. In *Underwater Technology, Proceedings of the 2000 International Symposium on*, pages 78–83, 2000.

[Lap06] Lionel Lapierre. Underwater robots part 1 : Current systems and problem pose. In Aleksandar Lazinica, editor, *Mobile Robots Towards New Applications*, chapter 16, pages 309–334. Advanced Robotic Systems International et pro Literatur Verlag, 2006.

[Lib03] Daniel Liberzon. *Swithcing in Systems and Control*. Birkhäuser, 2003.

[LLA+14] Adrien Lasbouygues, Lionel Lapierre, David Andreu, Josue Lopez Hermoso, Herve Jourde, and Benoit Ropars. Stable and reactive centering in conduits for karstic exploration. In *European Control Conference (ECC'14)*, 2014.

[LRP+15] Adrien Lasbouygues, Benoit Ropars, Robin Passama, David Andreu, and Lionel Lapierre. Atoms based control of mobile robots with Hardware-in-the-Loop validation. In *Intelligent Robots and Systems (IROS 2015), IEEE International Conference on*, 2015.

[LSV03] Nancy Lynch, Roberto Segala, and Frits Vaandrager. Hybrid i/o automata. *Information and Computation*, 185(1) :105–157, 2003.

[McC01] Patrick McCully. *Silenced Rivers : The Ecology and Politics of Large Dams*. Zed Books, 2001.

[MCC12] D. Maalouf, V. Creuze, and A. Chemori. State feedback control of an underwater vehicle for wall following. In *MED'12 : 20th Mediterranean Conference on Control and Automation*, 2012.

[MCJP10] Francesco Maurelli, Joel Cartwright, Nicholas Johnson, and Yvan Petillot. Nessie iv autonomous underwater vehicle wins the sauc-e competition. In *10th Inernational Conference on Mobile Robots and Competitions (ROBOTICA2010), Proceedings of*, 2010.

[MF13] Aaron Martinez and Enrique Fernández. *Learning ROS for Robotics Programming*. Packt Publishing, Septembre 2013.

[MKH98] Vikram Manikonda, P.S. Krishnaprasad, and James Hendler. Languages, behaviors, hybrid architectures, and motion control. In John Baillieul and Jan Camiels Willems, editors, *Mathematical Control Theory*, pages 199–226. Springer New York, 1998.

[MLN+12] Coralie Monpert, Michel Legris, Claire Noel, Benoit Zerr, and Jean Marc Le Caillec. Studying and modeling of submerged aquatic vegetation environments seen by a single beam echosounder. In *Proceedings of meetings on acoustics, 11th European Conference on Underwater Acoustics (ECUA 2012)*, 2012.

[MNS13] Alessandra Melani, Eric Noulard, and Luca Santinelli. Learning from probabilities : Dependences within real-time systems. In *Emerging Technologies and Factory Automation (ETFA'2013)*, 2013.

[Mod00] Modelica Association. *Modelica - A Unified Object-Oriented Language for Physical Systems Modeling*, 2000.

[MWE07] Mark Moline, Dana Woodruff, and Nathan Evans. Optical delineation of benthic habitat using an autonomous underwater vehicle. *Journal of Field Robotics*, 24(6) :461–471, 2007.

[Nil69] Nils John Nilsson. A mobile automaton : An application of artificial intelligence techniques. Technical Report 40, AI Center, SRI International, Mars 1969.

[NJP08] Kenza Najib, Hervé Jourde, and Séverin Pistre. A methodology for extreme groundwater surge predetermination in carbonate aquifers : Groundwater flood frequency analysis. *Journal of Hydrology*, 352(1–2) :1–15, 2008.

[Oce12] Oceaneering. *ROV Spectrum Datasheet*, Avril 2012. Disponible à l'adresse : `http://www.oceaneering.com/oceandocuments/brochures/rov/ROV-Spectrum.pdf`.

[Oce15] Deep Ocean. *Phantom T3 Specifications*, Janvier 2015. Disponible à l'adresse : `http://www.deepocean.com/pdf/t3.pdf`.

[PACGD14] Robin Passama, David Andreu, Didier Crestani, and Karen Godary-Dejean. Architectures de contrôle pour la robotique - approches et tendances. *Techniques de l'ingénieur Robotique*, base documentaire : TIB398DUO.(ref. article : s7791), 2014. Disponible à l'adresse : `http://www.techniques-ingenieur.fr/base-documentaire/electronique-automatique-th13/robotique-42398210/architectures-de-controle-pour-la-robotique-s7791/`.

[Pas10] Robin Passama. Contract : une méthodologie de conception et de développement d'architectures de contrôle de robots. Technical Report RR-10025, LIRMM, 2010.

[PBP00] Michel Poupart, P. Benefice, and Michel Plutarque. Subaquatic inspections of edf (electricite de france) dams. In *OCEANS 2000 MTS/IEEE Conference and Exhibition*, volume 2, pages 939–942, 2000.

[PFC14] Edson Prestes, Sandro Rama Fiorini, and Joel Carbonera. Core ontology for robotics and automation. In *Standardized Knowledge Representation and Ontologies for Robotics and Automation, Workshop on the*, pages 7–9, 2014.

[Pid06] Michael Pidwirny. The hydrologic cycle, 2006. `http://www.physicalgeography.net/fundamentals/8b.html`.

[PRP+12] Mario Prats, David Ribas, Narcís Palomeras, Juan Carlos García, Volker Nannen, Stephan Wirth, José Javier Fernández, Joan Beltrán, Ricard Campos, Pere Ridao, Pedro Sanz, Gabriel Oliver, Marc Carreras, Nuno Gracias, Raúl Marín,

and Alberto Ortiz. Reconfigurable auv for intervention missions : a case study on underwater object recovery. *Intelligent Service Robotics*, 5(1) :19–31, 2012.

[QGC+09] Morgan Quigley, Brian Gerkey, Ken Conley, Josh Faust, Tully Foote, Jeremy Leibs, Eric Berger, Rob Wheeler, and Andrew Ng. ROS : an open-source robot operating system. In *ICRA workshop on Open-source Software*, 2009.

[RLLA15] Benoit Ropars, Adrien Lasbouygues, Lionel Lapierre, and David Andreu. Thruster's dead-zones compensation for the actuation system of an underwater vehicle. In *European Control Conference (ECC'15)*, 2015.

[RM10a] Olena Rogovchenko and Jacques Malenfant. Composition and compositionality in a component model for autonomous robots. In Benoit Baudry and Eric Wohlstadter, editors, *Software Composition*, volume 6144 of *Lecture Notes in Computer Science*, pages 34–49. Springer Berlin Heidelberg, 2010.

[RM10b] Olena Rogovchenko and Jacques Malenfant. Handling hardware heterogeneity through rich interfaces in a component model for autonomous robotics. In Noriaki Ando, Stephen Balakirsky, Thomas Hemker, Monica Reggiani, and Oskar von Stryk, editors, *Simulation, Modeling, and Programming for Autonomous Robots*, volume 6472 of *Lecture Notes in Computer Science*, pages 312–323. Springer Berlin Heidelberg, 2010.

[RMT14] Anand Ramaswamy, Bruno Monsuez, and Adriana Tapus. Saferobots : A model-driven framework for developing robotic systems. In *Intelligent Robots and Systems (IROS 2014), 2014 IEEE/RSJ International Conference on*, pages 1517–1524, 2014.

[Rob10] Bluefin Robotics. *Bluefin-9 product sheet*, 2010. Disponible à l'adresse : `http://www.bluefinrobotics.com/assets/Downloads/Bluefin-9-Product-Sheet.pdf`.

[SACG11] Abusayeed Saifullah, Kunal Agrawal, Lu Chenyang, and Christopher Gill. Multi-core real-time scheduling for generalized parallel task models. In *Real-Time Systems Symposium (RTSS), 2011 IEEE 32nd*, pages 217–226, Novembre 2011.

[SL91] Jean-Jacques Slotine and Weiping Li. *Applied Nonlinear Control*. Prentice-Hall, 1991.

[SMKR11] Mac Schwager, Nathan Michael, Vijay Kumar, and Daniela Rus. Time scales and stability in networked multi-robot systems. In *Robotics and Automation (ICRA), 2011 IEEE International Conference on*, pages 3855–3862, 2011.

[Soe12] Peter Soetens. *The Orocos Component Builder's Manual : Open RObot COntrol Software : 2.6.0*, 2012.

[SPGA06] Daniel Simon, Roger Pissard-Gibollet, and Soraya Arias. Orccad, a framework for safe robot control design and implementation. In *1st national workshop on control architectures of robots : software approaches and issues CAR'06*, 2006.

[SSS14] Daniel Simon, Ye-Qiong Song, and Olivier Sename. Control and scheduling joint design. In Maryline Chetto, editor, *Real-time Systems Scheduling*, volume 2, pages 53–96. ISTE Editions, Septembre 2014.

[Ste01] David Stewart. Designing software components for real-time applications. In *Embeded System Conference*, 2001.

[Tam13] Tamboritha. *Light Work Class ROV - Saab SeaEye Cougar XT Datasheet*, Juillet 2013. Disponible à l'adresse : `http://tamboritha.com.au/tcwp/wp-content/uploads/2013/07/Datasheet-Light-Work-ClassROV.pdf`.

[Tec13] Rowe Technologies. *SeaPILOT Datasheet*, Décembre 2013. Disponible à l'adresse : `http://rowetechinc.com/wp-content/uploads/2013/12/LO-RES-SeaPILOT-300-600-1200-DVL-Brochure-27-Dec-2013.pdf`.

[TRC+10] Juan Marcos Toibero, Flavio Roberti, Fernando Auat Cheein, Carlos Soria, and Ricardo Carelli. Stable switching control of wheeled mobile robots. In Alejandra Barrera, editor, *Mobile Robots Navigation*, chapter 19, pages 379–400. InTech, 2010.

[Tre15] Deep Trekker. *Deep Trekker DTX2*, Avril 2015. Disponible à l'adresse : `http://deeptrekker.com/wp-content/uploads/2015/04/Deep-Trekker-DTX2.pdf`.

[Tri15] Tritech. *Super SeaKing Profiler Datasheet*, 2015. Disponible à l'adresse : `http://www.tritech.co.uk/media/products/dual-frequency-profiler-super-seaking-dfp.pdf`.

[UFH+12] Peter Ulbrich, Florian Franzmann, Christian Harkort, Martin Hoffmann, Tobias Klaus, Anja Rebhan, and Wolfgang Schröder-Preikschat. Taking control : Modular and adaptive robotics process control systems. In *Robotic and Sensors Environments (ROSE), 2012 IEEE International Symposium on*, 2012.

[Vid14] VideoRay. *PRO 4 ULTRA BASE*, 2014. Disponible à l'adresse : `http://www.videoray.com/images/specsheets/2014/2014_PRO4ULTRABASE_FINAL3.pdf`.

[VNE+00] Richard Volpe, Issa A. D. Nesnas, Tara Estlin, Darren Mutz, Richard Petras, and Hari Das. CLARAty : Coupled layer architecture for robotic autonomy. Technical report, NASA - JET PROPULSION LABORATORY, 2000.

[Wal50] William Grey Walter. An imitation of life. *Scientific American*, pages 42–45, Mai 1950.

[Wal09] Tobias Walter. Combining domain-specific languages and ontology technologies. In *Proceedings of the doctoral symposium at MODELS. Technical Report*, pages 34–39, 2009.

[WR09] Nicky Williams and Muriel Roger. Test generation strategies to measure worst-case execution time. In *Automation of Software Test, 2009. AST '09. ICSE Workshop on*, pages 88–96, 2009.

[WS05] David Watson and David Scheidt. Autonomous systems. *Johns Hopkins APL technical digest*, 26(4) :368–376, 2005.

[ZBK11] Michael Zolda, Sven Bünte, and Raimund Kirner. Context-sensitive measurement-based worst-case execution time estimation. In *Embedded and Real-Time Computing Systems and Applications (RTCSA), 2011 IEEE 17th International Conference on*, volume 1, pages 243–250, 2011.

[ZBP01] Wei Zhang, Michael Branicky, and Stephen Phillips. Stability of networked control systems. *Control Systems, IEEE*, 21(1) :84–99, 2001.

[ZL93] R. Zapata and P. Lepinay. Fast mobile robots in unstructured environments. *Lecture Notes in Computer Science*, 708 :94–105, 1993.

Exploration robotique de l'environnement aquatique : Les modèles au cœur du contrôle

Résumé :

Les robots sous-marins peuvent aujourd'hui évoluer dans des environnements complexes difficilement accessibles à l'Homme pour des raisons de coût ou de sécurité. Ils peuvent donc intervenir dans une grande variété de missions en environnement aquatique. Or, la complexité de ces milieux impose de doter le vecteur robotique d'une autonomie opérationnelle suffisante afin qu'il puisse mener sa mission à bien tout en préservant son intégrité. Cela nécessite de développer des lois de commande répondant aux spécificités de l'application. Ces lois de commande se basent sur des connaissances provenant de différentes disciplines scientifiques ce qui souligne l'interdisciplinarité inhérente à la robotique. Une fois la loi de commande développée, il faut implémenter le contrôleur sur le robot sous forme de logiciel de contrôle basé sur une architecture logicielle temps-réel.

Or la conception actuelle des lois de commande, sous forme de blocs "monolithiques", rend difficile l'évolution d'une loi de commande d'une application à l'autre, l'intégration de connaissances provenant d'autres disciplines scientifiques que ne maîtrisent pas forcément les automaticiens et pénalise son implémentation sur des architectures logicielles qui nécessitent la modularité. Pour résoudre ces problèmes nous cherchons à proprement séparer les différentes connaissances afin que chacune soit aisément manipulable, son rôle clair et que les relations établies entre les différentes connaissances soient explicites. Cela permettra en outre une projection plus efficace sur l'architecture logicielle. Nous proposons donc un nouveau formalisme de description des lois de commande selon une composition modulaire d'entités de base appelées Atomes et qui encapsulent les différents éléments de connaissance.

Nous nous intéressons également à l'établissement d'une meilleure synergie entre les aspects automatique et génie logiciel qui se construit autour de préoccupations communes telles que les contraintes temporelles et la stabilité. Pour cela, nous enrichissons nos Atomes de contraintes chargées de véhiculer les informations relatives à ces aspects temporels. Nous proposons également une méthodologie basée sur notre formalisme afin de guider l'implémentation de nos stratégies de commande sur un Middleware temps-réel, dans notre cas le Middleware ContrACT développé au LIRMM.

Nous illustrons notre approche par diverses fonctionnalités devant être mises en œuvre lors de missions d'exploration de l'environnement aquatique et notamment pour l'évitement de parois lors de l'exploration d'un aquifère karstique.

Mots-clés : Robotique sous-marine, exploration environnementale, modularité, logiciel de contrôle, contraintes temporelles, temps-réel

Robotic exploration of the aquatic environment : Models at the core of the control.

Abstract :

Underwater robots can nowadays operate in complex environments in a broad scope of missions where the use of human divers is difficult for cost or safety reasons. However the complexity of aquatic environments requires to give the robotic vector an autonomy sufficient to perform its mission while preserving its integrity. This requires to design control laws according to application requirements. They are built on knowledge from several scientific fields, underlining the interdisciplinarity inherent to robotics. Once the control law designed, it must be implemented as a control Software working on a real-time Software architecture.

Nonetheless the current conception of control laws, as "monolithic" blocks, makes difficult the adaptation of a control from an application to another and the integration of knowledge from various scientific fields which are often not fully understood by control engineers. It also penalizes the implementation of control on Software architectures, at least its modularity and evolution. To solve those problems we seek a proper separation of knowledge so that each knowledge item can be easily used, its role precisely defined and we want to reify the interactions between them. Moreover this will allow us a more efficient projection on the Software architecture. We thus propose a new formalism for control laws description as a modular composition of basic entities named Atoms used to encapuslate the knowledge items.

We also aim at building a better synergy between control and software engineering based on shared concerns such as temporal constraints and stability. Hence we extend the definition of our Atoms with constraints carrying information related to their temporal behavior. We propose as well a methodlogy relying on our formalism to guide the implementation of control on a real-time Middleware. We will focus on the ContrACT Middleware developped at LIRMM.

Finally we illustrate our approach on several robotic functionnalities that can be used during aquatic environments exploration and especially for wall avoidance during the exploration of a karst aquifer.

Keywords : Underwater robotics, environment exploration, control modularity, control software, temporal constraints, real-time