Contrôle d'exécution dans une architecture hiérarchisée pour systèmes autonomes

Frédéric Py

Thèse

préparée au

Laboratoire d'Analyse et d'Architecture des Systèmes du CNRS

en vue de l'obtention du

Doctorat de l'Université Paul Sabatier de Toulouse
Soutenue le 20 Octobre 2004
Spécialité : Robotique

par

Frédéric Py

Contrôle d'Exécution dans une Architecture Hiérarchisée pour Systèmes Autonomes

---

Soutenance le devant le jury :

Saddek Bensalem	**VERIMAG, Grenoble**	Rapporteur
Raja Chatila	**LAAS-CNRS, Toulouse**	Président du jury
Félix Ingrand	**LAAS-CNRS, Toulouse**	Directeur de thèse
Guy Juanole	**LAAS-CNRS, Toulouse**	Examinateur
David Powell	**LAAS-CNRS, Toulouse**	Examinateur
Eric Rutten	**INRIA Futurs, Villeneuve d'Ascq**	Rapporteur

LAAS-CNRS
7, Avenue du Colonel Roche
31077 Toulouse Cedex 4

A mon père ...

Remerciements

Il est maintenant temps de remercier toutes les personnes qui m'ont accompagnées, supportées ou encore soutenues durant cette thèse. La liste ne peut malheureusement être exhaustive et je m'excuse par avance auprès de ceux qui n'y seront pas explicitement nommés. Qu'ils sachent toutefois que je ne les oublie pas pour autant.

Je commence donc par remercier l'ensemble des membres de mon jury de m'avoir fait l'honneur d'évaluer mon travail. Parmi ceux-ci je tiens à remercier tout particulièrement Raja Chatila, chef du groupe RIA où j'ai sévi durant ces 4 années. J'ai pu apprécier durant cette période une personne qui, à mes yeux, allie à la perfection une ouverture scientifique indéniable à un humour teinté d'un cynisme délicieux.

Il est impossible d'oublier ici Félix Ingrand : mon directeur de thèse. Au delà d'un encadrement que j'ai pu apprécier comme excellent – que ce soit de manière relative ou absolue – j'ai découvert en lui des qualités humaines et une patience qui seront toujours à son honneur. Rien que le fait de m'avoir supporté tout au long de ces années est un exploit.

Lors de la rédaction de ce manuscrit plusieurs personnes ont été particulièrement présentes pour m'aider à rendre ce document "lisible". On retrouve ici Félix mais aussi Etienne Roblet, Nicolas Do Huu, Guillaume Infantes et Maxime Cottret. Je les remercie tous pour avoir effectué cette relecture et détecté les fautes impardonnables et autres phrases douteuses présentes dans ce manuscrit.

Enfin j'ai une pensée pour toutes les personnes que j'ai eu le plaisir de croiser durant cette période et qui m'ont permis de moins me prendre au sérieux. Parmis les membres du LAAS, il y a les deux sympathique fêtards que sont Guillaume et Stéph que je considére comme des frêres de cœur. On y trouve aussi Nico pour ses tournois de pokers et discussions autour d'un verre ... ou deux. Léo et ses appartements toujours euh ... différents ainsi que sa bonne humeur. Alex, Max et Séb m'ont permis d'avoir toujours un compagnon pour prendre un café et ainsi avoir moins l'impression d'être un caféïnomane. Finalement, la french délégation – représentée par Will et Ben – m'a permis de dédramatiser ma première conférence et de voir le positif dans un désastre cataclismique. Pour les personnes extérieures au LAAS, je ne peux oublier la bande de rôlistes que sont Etienne, Nicolas et Boris pour toute les traitrises et autres "surprises" qu'ils m'ont réservées. Il y a aussi Thony pour ses khebabs et délires poétiques et je

ne peux oublier Ronan, le “pauv’ petit chouchou, dressed like a Pazuzu”, qui sera là jusqu’aux enfers.

Il reste encore beaucoup de personnes à remercier mais la place et l’inspiration me manquent. Je finis donc ici par remercier ma mère et ma sœur pour avoir cru en moi, même si je n’étais pas toujours facile à vivre.

Table des matières

Table des figures

Chapitre I

Problématique & état de l'art

Il y a un besoin grandissant pour une autonomie de haut niveau dans des systèmes temps réels complexes tels que les robots ou les satellites. Afin d'offrir des réponses à cette problématique, de nombreux travaux de recherche ont porté sur la définition d'architectures logicielles pour les systèmes autonomes. Toutefois, les domaines où le besoin d'autonomie est le plus grand sont, en général, aussi des domaines où des garanties du bon fonctionnement du système sont nécessaires. Afin de présenter cette problématique, nous posons ici le contexte en donnant une définition des systèmes autonomes et les architectures existantes dans le domaine. A partir de ces bases, nous présenterons la problématique liée à la fiabilité de tels systèmes et les solutions déjà proposées dans la littérature. Ce tour d'horizon ainsi que l'analyse qui l'accompagne nous permettront enfin d'introduire l'approche que nous avons choisie.

I.1 Les systèmes autonomes

I.1.1 Qu'est-ce qu'un système autonome ?

Un système autonome peut être vu comme une entité agissant dans un environnement fortement variable avec une intervention humaine réduite. Dans cette définition, on considère deux axes symbolisant les besoins liés à l'autonomie :

– La capacité d'agir avec une intervention humaine réduite : les objectifs fournis au système sont généralement de très haut niveau avec un horizon temporel lointain. Par conséquent, le système doit être capable de prendre toute décision nécessaire au bon déroulement de sa mission et à la réalisation des objectifs visés.

– La variabilité de l'environnement : elle menace en permanence le plan de notre système. Il doit donc exister des mécanismes offrant une robustesse à cette menace.

Un système ne satisfaisant qu'une seule de ces deux propriétés ne peut être considérée comme autonome. Par exemple, un robot industriel agit avec une intervention humaine quasi nulle. Par contre l'environnement dans lequel il évolue et les tâches qu'il doit effectuer sont parfaitement connues. Ceci simplifie grandement la problématique et les logiciels embarqués se limitent à des automates déterministes sans réelle prise de décision. Les actions à effectuer ainsi que leur séquence sont connues a priori et ne seront pas remises en cause directement par le système. De même, les systèmes téléopérés évoluent dans un environnement dont le système ne connaît pas a priori l'évolution mais le choix des actions ainsi que leur ordonnancement est laissé à un opérateur humain. Ainsi la prise de décision est quasi inexistante pour ce type de système et il ne fait qu'exécuter les commandes de bases initiées par un agent externe.

Pour pouvoir agir correctement en tenant compte de ces deux aspects, un système autonome va se trouver avec des besoins nouveaux qu'on ne trouvera pas dans des systèmes plus classiques.

Premièrement de tels systèmes auront des capacités décisionnelles adaptées à leur haut niveau d'autonomie. Les objectifs donnés ne pouvant être exécutés directement, ce système doit être capable de déduire les actions à effectuer ainsi que leur ordre pour atteindre les buts fixés. Afin de traiter ce type de problème, de tels systèmes embarquent des outils complexes prenant en compte le temps, les ressources ou encore l'incertitude du monde. De nombreux travaux d'intelligence artificielle offrent de telles possibilités. On trouve par exemple des planificateurs symboliques intégrant le temps et les ressources capables de produire des plans avec exécution parallèle des actions.

Le second point porte sur la robustesse de notre système vis-à-vis de l'environnement. Le terme de robustesse a longtemps été exploité en robotique pour exprimer la capacité qu'a un système d'effectuer sa tâche malgré les situations adverses. Dans [Lussier 04], les auteurs tentent d'offrir une base plus formelle à ce terme en le différenciant des autres concepts liés à la sûreté de fonctionnement. Ainsi ils définissent chacun de ces deux concepts de la façon suivante :

La robustesse est la capacité de rendre un service correct dans une situation

adverse non prévisible issue de l'environnement (obstacle inattendu, porte fermée, etc.).

La tolérance aux fautes est la capacité de rendre un service correct indépendamment de fautes affectant les éléments composant le système (mauvais fonctionnement d'un senseur, faute logicielle, ...).

Cette définition permet de bien différencier la robustesse et la tolérance aux fautes. Toutefois, les techniques utilisées pour chacun restent assez proches ; par exemple, on trouvera fréquemment des redondances fonctionnelles ou des mécanismes de reprise d'exécution dans les deux cas.

On ajoute qu'un tel système, pour être exploitable, doit respecter les propriétés suivantes :

Réactivité De par la nature même de l'environnement, le système doit être capable de réagir à une situation non prévue dans un délai garantissant l'exécution sûre de la tâche qu'il doit accomplir.

Sûreté Les systèmes autonomes sont utilisés dans des situations critiques : ils peuvent être en interaction avec des humains (robots guides de musée, robots personnels, ...) ou encore intervenir dans des domaines sensibles (centrales nucléaires, sauvetage, ...). Il est nécessaire d'offrir des garanties quant à leur bon fonctionnement. Ceci est d'autant plus vrai que l'autonomie est grande. En effet, la prise de décision étant à la charge du système, il devient dès lors critique de pouvoir certifier que les actions décidées ne sont pas une menace quant à son intégrité ainsi que celle de son environnement.

Programmabilité, traçabilité, ... Ces propriétés sont moins critiques et/ou spécifiques aux systèmes autonomes. Toutefois elles sont nécessaires lors de la phase de développement et d'intégration de ce type de plate-forme. On y trouve une multiplicité d'outils exploitant des formalismes et techniques très éloignés les uns des autres. Par conséquent, l'intégration de l'ensemble est difficile et doit être supportée par des outils durant les différentes phases de la conception du système.

I.1.2 Tour d'horizon des architectures

Il est intéressant de constater que c'est la communauté robotique qui offre le plus de travaux sur les architectures génériques pour systèmes autonomes. Plusieurs études ont été faites afin d'offrir une plate-forme logicielle générique

et évolutive intégrant les capacités décisionnelles suffisantes pour que le système puisse agir de façon autonome dans son environnement.

Nous présentons ici deux classes d'architectures représentatives de ce qui a pu être proposé. Ce bref aperçu nous permettra d'avancer certains avantages des architectures hiérarchisées présentées dans la section I.2.

Les architectures comportementales

Les architectures comportementales ont la spécificité de ne pas offrir – du moins pas de façon centralisée – un raisonnement prédictif et un modèle global du monde. L'idée est de diviser le système en un ensemble de comportements relativement simples. Le comportement global du système face à une situation n'est pas explicitement planifié, mais résulte de la composition de ces comportements.

Chaque niveau de comportement réagit aux entrées, données par l'environnement, afin de produire une sortie correspondant au comportement associé indépendamment des autres niveaux. Un arbitrage est effectué sur les sorties dans le cas où des décisions de deux niveaux seraient conflictuelles.

Parmi les travaux les plus notables dans ce domaine, on trouve ceux de Brooks [Brooks 91] qui s'appuient principalement sur son architecture de subsomption [Brooks 86], représentée dans la figure I.1. On décompose ici le système en plusieurs niveaux avec chacun un ensemble fixe de modules dont le comportement est similaire à un automate à états finis. La politique de gestion de conflits fait en sorte que les niveaux les plus élevés peuvent bloquer les entrées ou inhiber les sorties des niveaux qui lui sont inférieurs. Ainsi, les niveaux les plus bas correspondront à des actions de bas niveau (par exemple évitement basique des obstacles) alors que ceux de plus haut niveau ajouteront des fonctionnalités plus complexes.

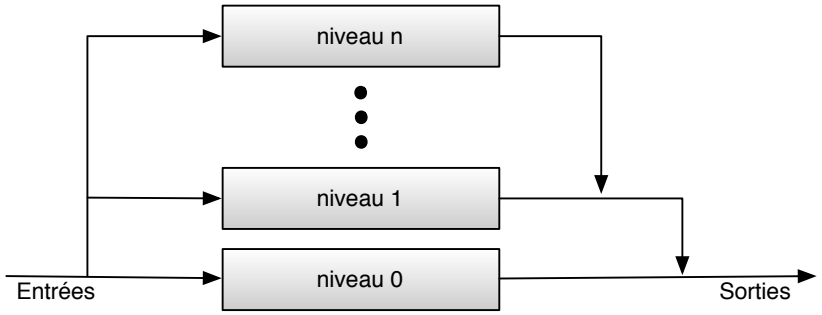


Fig. I.1 – L'architecture de subsomption

Cette approche a obtenu un grand succès grâce à son principe de décomposition des comportements. Des tâches d'une grande complexité peuvent émerger à partir

d'un ensemble de comportements plus basiques. Elle a toutefois certaines limitations :

– Si ce système est très évolutif (il suffit d'ajouter un nouveau niveau incluant la nouvelle fonctionnalité), sa flexibilité est faible : dans l'architecture de subsomption, il faut tenir compte de tous les niveaux inférieurs pour développer un nouveau niveau. Ainsi l'insertion d'un nouveau niveau au bas de la hiérarchie entraîne la modification de tous les niveaux supérieurs.

– Quand la complexité augmente, les interactions entre comportements font de même. Il devient alors difficile de pouvoir prédire le comportement émergeant.

– Les comportements sont efficaces quand la donnée qui leur est nécessaire est directement disponible. Par contre si celle-ci est plus complexe et évoluée, il devient difficile de la mettre en œuvre dans ce type d'architecture.

– L'absence de réel pouvoir de décision par anticipation, oblige à expliciter celle-ci lors de la conception. Ce problème est d'autant plus mis en évidence que le niveau exhibe un comportement évolué.

Les architectures centralisées

Les architectures centralisées sont structurées autour d'un élément central qui sert généralement à la synchronisation et la communication entre les autres composants. Plusieurs d'entre elles adoptent le paradigme du tableau noir (*blackboard*). Le tableau noir permet ici de coordonner l'interaction entre les divers agents. Ce genre d'organisation s'appuie sur les postulats suivants [Hayes-Roth 85] :

– Tous les éléments de la solution générés durant la résolution d'un problème sont enregistrés dans une base de données globale et structurée, appelée tableau noir.

– Ces éléments sont générés et enregistrés par des processus indépendants, appelés les sources de connaissance.

– Pour chaque cycle de résolution, un mécanisme d'arbitrage choisit une source de connaissance parmi celles qui peuvent être activées, pour exécuter son action.

Une autre approche centralisée fut adoptée par R. Simmons avec l'architecture TCA[1][Simmons 94](figure I.2). Ici, le contrôle est centralisé, mais les données sont distribuées entre les processus. Chaque agent peut contenir un processus de perception, d'action et de prise partielle de décision, pour tout ce qui peut être traité localement et qui ne dépend pas des autres agents. Ils communiquent entre eux via le contrôleur central. Ces messages peuvent être des données ou des demandes de services que le contrôleur envoie aux agents concernés.

[1]TCA : Task control architecture

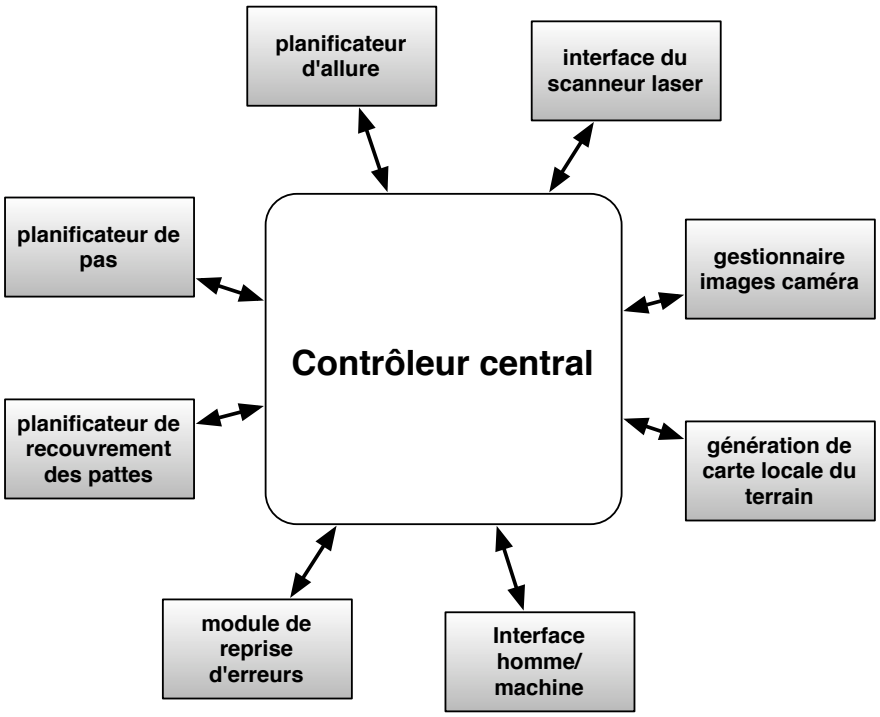


FIG. I.2 – L'architecture centralisée TCA

Ces architectures sont en général fortement évolutives et flexibles grâce à leur structure non hiérarchisée [Tigli 93]. Le module central a un fonctionnement clairement défini définissant ainsi un cadre pour la conception et la combinaison des agents. Par contre, l'élément central est un goulot d'étranglement qui limite les possibilités d'utilisation avec un grand nombre de composants. De plus la centralisation entraîne une mise à plat des composants dont l'exécution est cadencée par le contrôleur central. Ainsi, l'intégration de composants complexes et lents devient délicate quand on souhaite garder une certaine réactivité.

Ces deux architectures sont intéressantes quand on souhaite avoir un système qui réagit à l'environnement avec un pouvoir d'anticipation relativement faible. Par contre, dès lors que le besoin de planifier se présente, elles deviennent trop limitatives. Les architectures hiérarchisées ont été créées afin de répondre à ce besoin.

I.2 Les architectures hiérarchisées.

Les architectures hiérarchisées – souvent appelées architectures à 3 couches – ont été créées afin de permettre l'intégration de composants décisionnels avec une grande complexité algorithmique tout en laissant une certaine réactivité au système. Elles s'appuient sur le paradigme de base percevoir/planifier/agir en y ajoutant l'hypothèse que les besoins de réactivité diminuent quand le niveau d'abstraction augmente. Une structuration classique est représentée dans la figure I.3. Ici, les composants de plus bas niveaux – "collant" à la couche physique – vont avoir des contraintes temps réels fortes avec une vision très locale de l'environnement (ils seront en général consacrés à un capteur/effecteur spécifique du système) alors qu'en montant dans la hiérarchie, nous allons rencontrer une complexité plus grande mais aussi une vision plus globale du monde, aidant à la prise de décision.

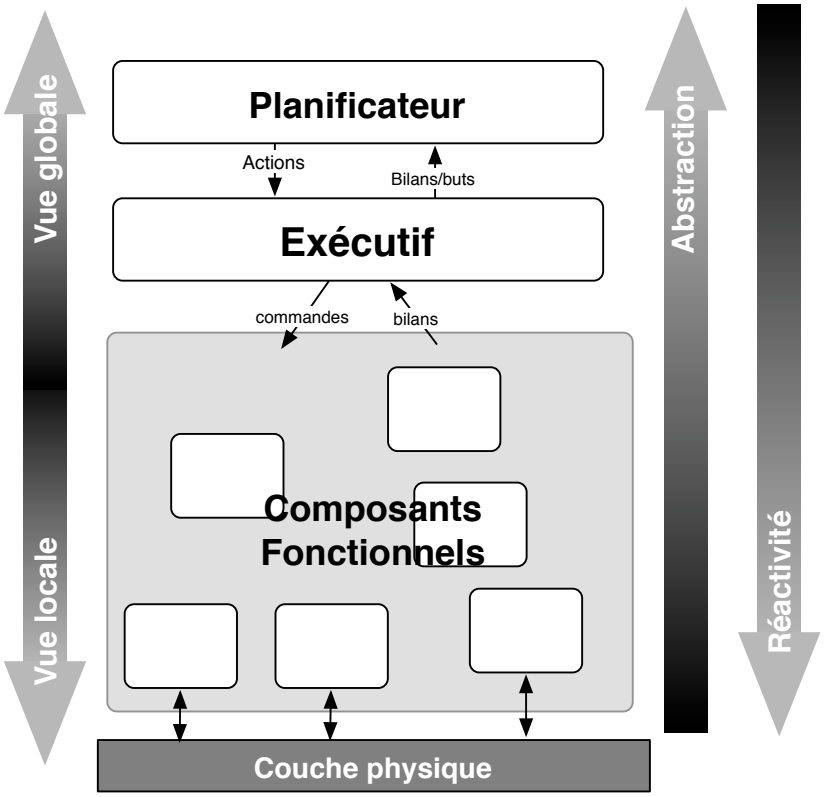


Fig. I.3 – Une architecture hiérarchisée

Cette hiérarchisation, où l'on trouve un lien entre niveau d'abstraction et

complexité algorithmique, permet d'intégrer des composants décisionnels complexes tout en gardant une certaine réactivité. Ce type d'architecture est particulièrement adapté à l'intégration de planificateurs symboliques. Placé au sommet de la hiérarchie, ce dernier aura une vision globale du monde – nécessaire pour produire un plan correct – tout en ayant un fort niveau d'abstraction qui compensera la complexité liée à ce type de logiciels. La réactivité est assurée par des composants de plus bas niveau. Ces propriétés permettent de concevoir des systèmes qui ont une capacité à se projeter dans le futur tout en conservant un pouvoir d'adaptation à l'environnement suffisant.

La majorité des architectures de ce type sont décomposées en trois niveaux (par exemple CIRCA [Musliner 93], 3T, ATLANTIS[Gat 97] ainsi que l'architecture LAAS [Alami 98]) : le décisionnel, l'exécutif et le fonctionnel. On constate toutefois que le rôle du niveau exécutif varie fortement d'une architecture à l'autre. Afin de rester sur un plan général nous décomposerons cette hiérarchie en deux grandes couches :

La couche décisionnelle : située au sommet de la hiérarchie, elle implémente les fonctions délibératives du système. On y trouve généralement deux principaux composants :
- *Le planificateur* qui cherche la séquence d'actions à effectuer afin d'atteindre un but donné. C'est là que l'on trouve les informations sur le futur possible du système.
- *L'exécutif* permet la connexion entre le planificateur et la couche fonctionnelle. Il a pour rôle de décomposer les actions données de haut niveau en commandes exécutables, effectue le suivi de l'exécution et, en cas de problème, essaye de trouver une solution alternative ou informe le planificateur de l'échec.

La couche fonctionnelle : située à l'interface entre la couche matérielle et les composants de plus haut niveau, elle gère les fonctions de base du système. On y trouve en particulier ses fonctions sensorimotrices, de traitement ainsi que les boucles de contrôle. Généralement chaque fonctionnalité est encapsulée dans des composants logiciels indépendants regroupant l'ensemble des services associés à une fonction. Par exemple, un module centralise les fonctions de bases associées à la caméra embarquée sur le système (initialisation, prise de vue, réglages de l'optique, ...) alors qu'un autre permet de faire de la reconnaissance et du suivi d'objets dans un flot d'images.

On distingue deux principaux flux d'information dans ce type d'architecture :

Le flot de données permet généralement la remontée des informations liées aux perceptions du système. Ces données deviennent de plus en plus abstraites au fur et à mesure qu'elles montent dans la hiérarchie. En effet, chaque composant capturant une donnée va la traiter afin d'en extraire l'information pertinente et la rendre interprétable par les composants clients.

Par exemple la paire d'images perçue par le banc stéréo est transmise pour être traitée par le module de stéréo corrélation afin d'en extraire les informations 3D. Celles-ci transitent vers un module qui intègre ces informations dans un modèle numérique du terrain. Ce dernier est transformé en une carte d'accessibilité du terrain exploitée finalement par un module de planification de chemin. On voit ici que l'information est de plus en plus abstraite au fur et à mesure de sa progression au sein de l'architecture. L'enrichissement sémantique désiré s'accompagne de la perte d'une partie de l'information initiale.

Le flot de contrôle est exploité pour le contrôle des commandes et activités nécessaires à l'exécution des tâches du plan fourni par les composants de plus haut niveau. On y trouve les demandes de lancement de services et autres activités et, en retour, les bilans de ces mêmes activités et autres signaux permettant de contrôler leur bonne exécution.

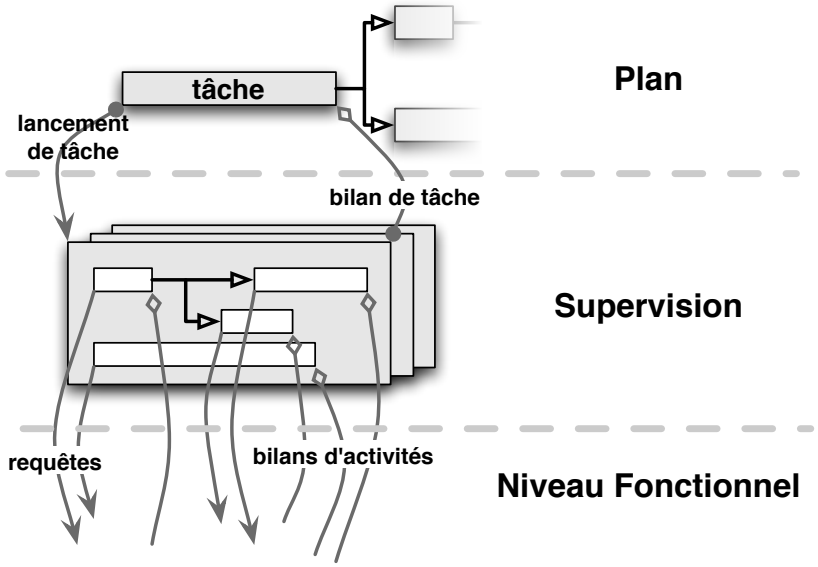


Fig. I.4 – Flot de contrôle dans une architecture hiérarchisée

Dans la figure I.4, on voit le lien entre l'abstraction des actions et le

flot de contrôle. Durant l'exécution du plan le lancement d'une tâche de haut niveau va être demandé au superviseur. Ce dernier choisit un des moyens possibles de l'exécuter qui va lui-même déclencher le lancement de diverses activités au niveau fonctionnel. Les bilans de ces activités vont ensuite être reçus par le superviseur. A la fin de l'exécution de cette tâche, un bilan est envoyé à l'exécutif de plan qui peut ainsi continuer à dérouler le plan.

Ainsi, on se trouve face à une abstraction des événements liés à l'abstraction des actions. Un événement de haut niveau – que ce soit une demande de lancement ou un bilan – correspond généralement à un ensemble d'événements de plus bas niveau chaque composant jouant un rôle de décomposition pour le flot descendant et d'abstraction pour le flot montant.

Il existe un grand nombre de propositions d'architectures hiérarchisées. Nous nous limitons ici à en présenter trois : l'architecture LAAS (I.2.1), CLARAty (I.2.2) et enfin IDEA (I.2.3).

I.2.1 L'architecture LAAS

Cette architecture est assez proche de la forme classique d'une architecture hiérarchisée. Toutefois, elle se distingue par la volonté d'y intégrer des outils offrant une cohérence globale du système [Alami 98]. Cette architecture a servi de base pour notre travail, nous nous limitons donc ici à une présentation générale. Une présentation plus détaillée sera donnée dans le chapitre II.

Comme l'indique la figure I.5, elle est composée de trois niveaux. Chacun d'eux possède ses propres contraintes temporelles et ses propres représentations de l'état du système. Ces niveaux sont :

– ***Le niveau décisionnel :*** Ce niveau a une structure relativement classique. Les outils exploités ici sont open-PRS [Ingrand 96], pour la supervision, et le planificateur symbolique IxTeT pour la planification qui inclut depuis [Lemai 04] le contrôle temporel de l'exécution de plan.

– ***Le niveau fonctionnel :*** On trouve ici les fonctions sensorimotrices, les fonctions de traitement ainsi que les boucles de contrôle (navigation, traitement d'images, capteurs, ...) du système. Chacune de ces fonctions est encapsulée dans un module généré par l'outil GenoM [Fleury 94]. Chaque module offre un ensemble de services, liés à une fonctionnalité donnée, accessibles aux clients via des requêtes. Les algorithmes sont décomposés en unités de code insécables appelées *codels*.

– ***Le niveau de contrôle d'exécution :*** Ce niveau est la particularité de cette architecture. Situé entre les deux niveaux présentés ci-dessus, il contrôle le bon déroulement de l'exécution. Il va en particulier vérifier si les actions demandées ne mènent pas le système dans un état incohérent. C'est dans cette couche que se trouve la contribution présentée dans ce mémoire.

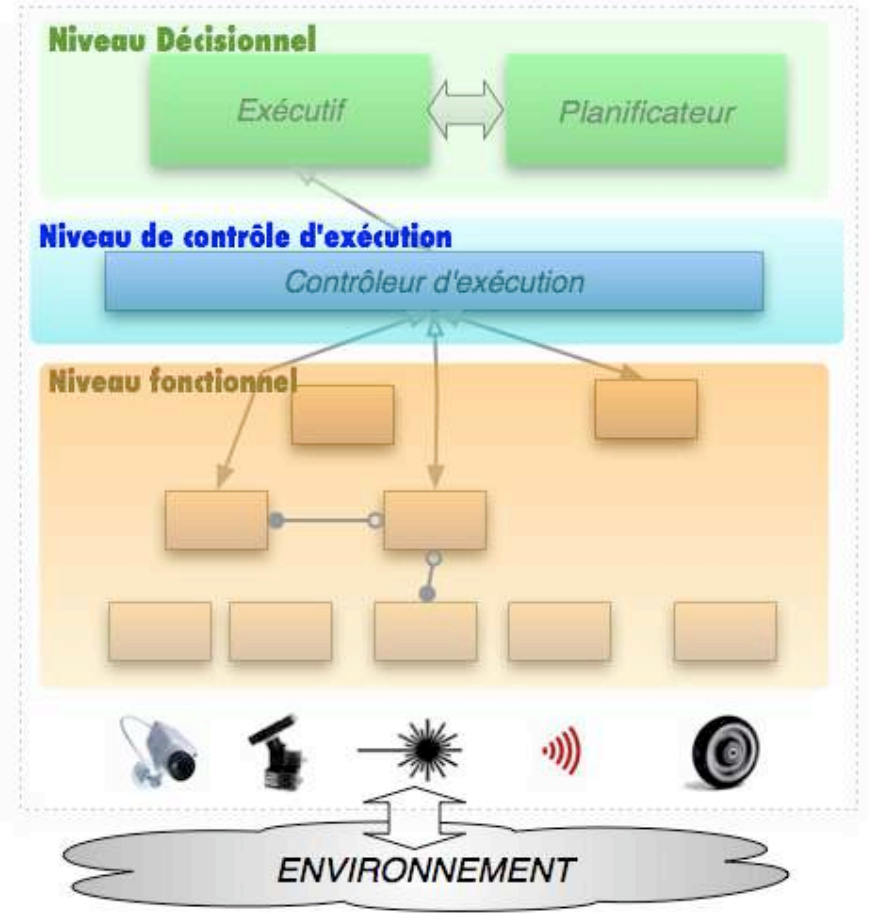


Fig. I.5 – L'architecture LAAS

Dans cette architecture, la mise en place d'outils facilitant le développement et l'intégration a toujours été une ligne directrice [Alami 00]. La spécification des comportements aux différents niveaux s'appuie sur des bases formelles (par exemple IxTeT) ou semi formelles (GenoM). L'intégration d'un système de contrôle d'exécution cherche à pallier les problèmes que nous présenterons plus en détail dans la section I.3.

I.2.2 L'architecture CLARAty

L'architecture CLARAty [Volpe 01] se propose de différencier le niveau d'abstraction du niveau de décision. En effet, les autres architectures considèrent plus ou moins que ces deux aspects sont relativement liés. Ici l'architecture explicite le niveau d'abstraction en le plaçant sur un autre axe que le niveau décisionnel (voir figure I.6). On se retrouve ainsi avec deux plans correspondant aux couches que nous avons définies :

1. *Le plan fonctionnel :* sur ce plan, le niveau d'abstraction est donné par une hiérarchie objet. Au plus haut niveau, on trouve des interfaces abstraites pour un type de service donné. Ces interfaces sont exploitées par les implémentations des fonctions se trouvant au niveau le plus faible d'abstraction.
2. *Le plan décisionnel :* ici l'abstraction se fait de façon plus classique pour ce type d'architecture. C'est une abstraction des actions : au sommet se trouvent les buts globaux qui sont raffinés jusqu'à des actions de base au fur et à mesure qu'on descend sur ce plan.

Le principal intérêt de cette approche est d'exploiter des paradigmes objets dans le système afin d'en augmenter la fiabilité globale. Dans la figure I.6 on voit qu'un composant avec un niveau d'abstraction assez élevé du plan décisionnel peut communiquer avec une interface de haut niveau du plan fonctionnel. Ceci permet d'aller chercher directement dans la couche fonctionnelle des données qui seront utiles pour la planification. Dans une hiérarchie classique, la même donnée aurait dû transiter par d'autres composants ce qui multiplie l'effort de mise en œuvre et les risques de faute.

Par contre, dans [Nesnas 03], on constate que la structuration orientée objet de la couche fonctionnelle met en évidence de nouvelles difficultés. En effet, ce choix entraîne un effort encore plus important sur l'unification des interfaces. Cet aspect existe déjà dans des systèmes modulaires tels que GenoM, mais est amplifié ici par la hiérarchie d'abstraction de classes imposée par cette architecture. Ainsi, si deux modules CLARAty offrent le même service, ils doivent nécessairement avoir une interface commune. Ceci est d'autant plus important que, comme nous l'avons indiqué précédemment, cette interface abstraite sera exploitée par la couche décisionnelle.

Cette unification est louable dans le sens où elle simplifie l'intégration globale, mais elle est loin d'être triviale. Deux composants offrant un service similaire, qui devraient donc avoir une interface commune, n'ont pas forcément les mêmes entrées et sorties. A partir de là, comment offrir une interface qui reste générique pour ce type de service sans qu'elle soit trop abstraite pour être exploitable ?

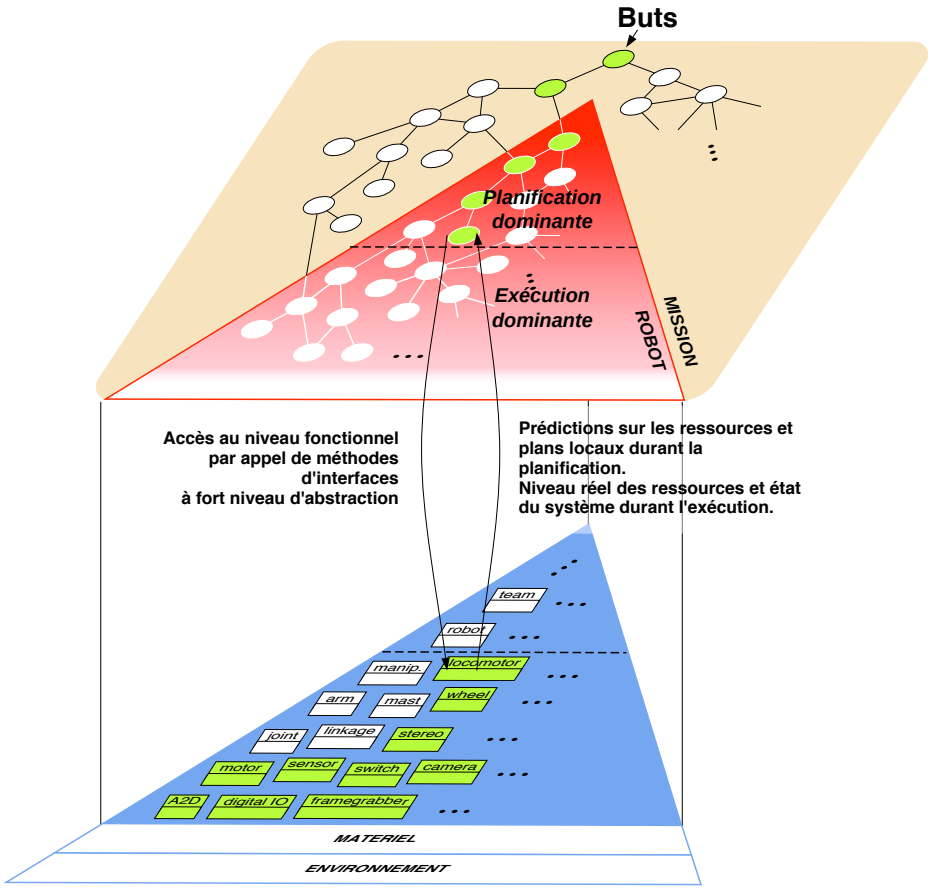


FIG. I.6 – L'architecture CLARAty

Les derniers travaux sur cette architecture [Bualat 04] semblent d'ailleurs fortement centrés sur cet aspect : leur objectif actuel est d'offrir une libraire de base assez générique pour être indépendante de la plate-forme physique sur laquelle elle sera exploitée.

I.2.3 IDEA : une architecture multi-agents

L'architecture IDEA [Muscettola 02], a été créée afin de mieux gérer les problèmes liés à la multiplicité de modèles dans les architectures classiques. Dans l'architecture *Remote Agent*[Muscettola 98], une incohérence entre le modèle exécutif et celui du module de détection de faute et reconfiguration (FDIR) est difficile à détecter car les deux composants n'exploitent pas le même formalisme de

représentation. D'un côté l'exécutif s'appuie sur l'*Execution Support Language*, un langage procédural permettant de spécifier la décomposition des tâches de haut niveau en actions de base. Le FDIR s'appuie sur Livingstone qui est plutôt basé sur un modèle des composants de bas niveau. La figure I.7 donne un exemple de modèle Livingstone. On y voit un modèle donnant la structure du système avec les dépendances entre composants et, pour chaque composant, un modèle qualitatif donnant les états possibles du système en y incluant l'état de panne (Stuck). Dans le rapport technique [Bernard 00], la section portant sur les leçons apprises

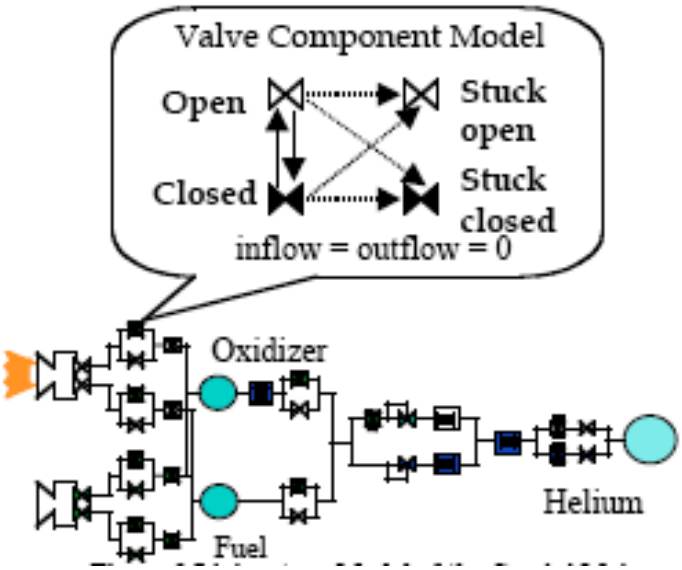


FIG. I.7 – Modèle Livingstone pour le moteur principal de Cassini

durant les expérimentations de *Remote Agent* met en évidence les problèmes liés à la multiplicité des modèles. En particulier, la modification d'un modèle donné a des répercussions sur l'ensemble du système et, si elle n'est pas intégrée dans les autres composants, des incohérences peuvent émerger.

Pour faciliter cette validation et limiter les risques d'incohérence, le choix a été fait d'unifier les modèles des divers composants de l'architecture. L'architecture IDEA est ainsi composée d'un ensemble d'agents – s'appuyant sur le même formalisme – exécutés par une machine virtuelle qui gère la communication et la synchronisation de ceux-ci. Chaque agent gère ici une partie de la planification avec une abstraction et un horizon plus ou moins grand. Les agents de plus bas niveau auront ainsi un horizon très court (planification réactive) avec un niveau d'abstraction faible, alors que ceux de plus haut niveau auront un horizon à long terme avec un niveau d'abstraction élevé. Le formalisme utilisé ici s'appuie sur

des *timelines* – illustrées par la figure I.8 – qui décrivent l'évolution des variables du système.

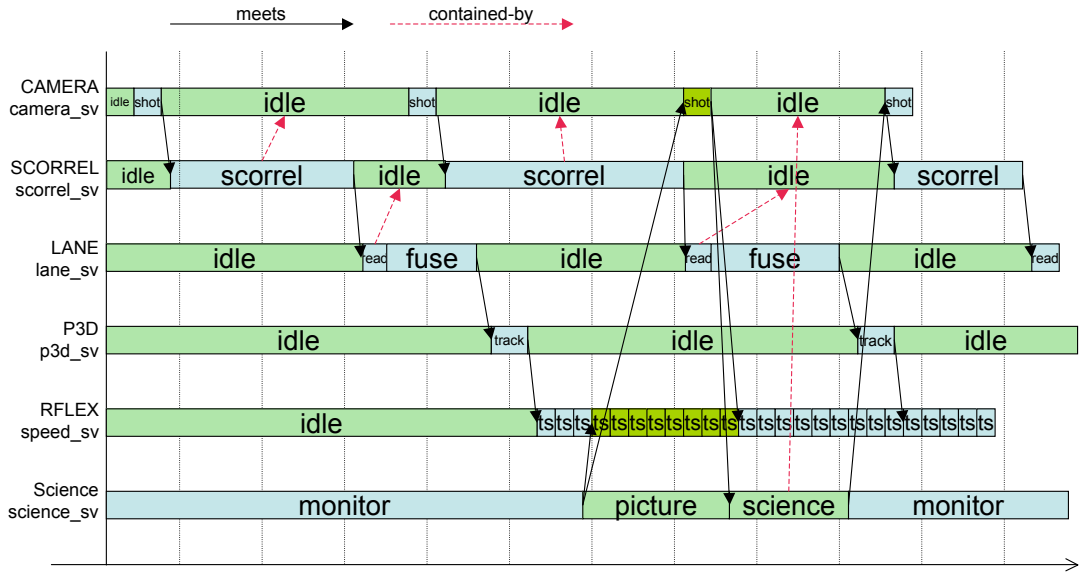
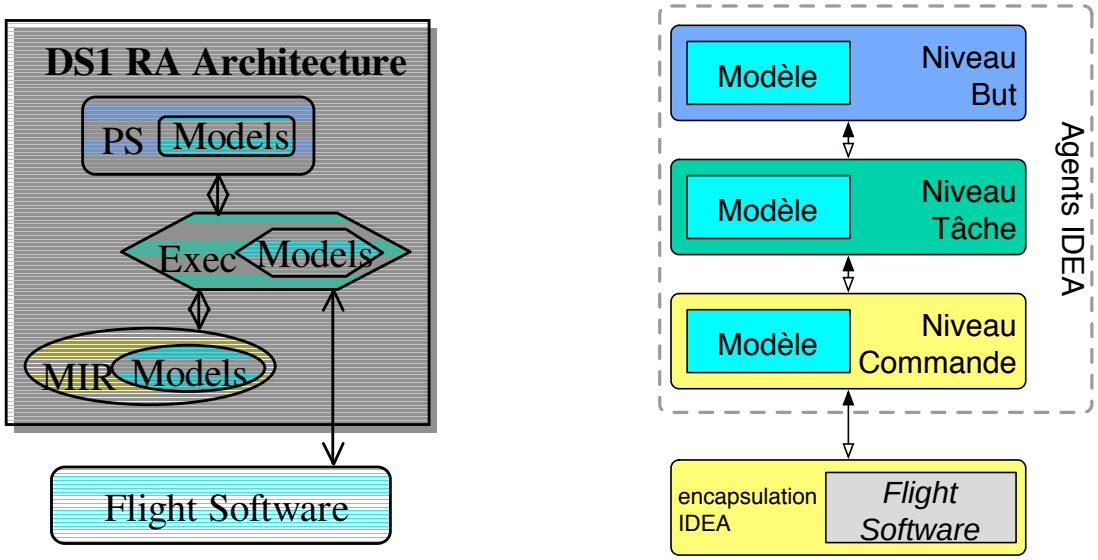


FIG. I.8 – Exemple de timelines IDEA

La synchronisation entre deux agents se fait via des *timelines* partagées. Par exemple, un agent qui a besoin que le robot soit à la position (x, y) va introduire cette nouvelle contrainte sur la *timeline* correspondante. L'agent qui supervise la navigation va ensuite planifier les actions nécessaires pour que ce but soit atteint. La synchronisation de l'ensemble est assurée par la machine virtuelle qui maintient la cohérence des *timelines* partagées.

Même si l'approche présentée ici semble plus générale que celle des architectures hiérarchisées classiques, nous avons fait le choix de présenter IDEA dans cette section. En effet, comme l'illustre la figure I.9, les auteurs se limitent, pour l'instant, à reproduire la structuration de Remote-Agent en exploitant les agents IDEA en lieu et place des anciens composants. Qui plus est, la corrélation qu'on trouve entre l'horizon de planification et le niveau d'abstraction de l'agent correspond bien au principe de base de ce type d'architecture.

Le principal intérêt que l'on peut trouver dans cette architecture est l'unification des modèles. Les auteurs mettent en particulier en avant le fait que ceci doit pouvoir permettre de faire une validation du fonctionnement global du système en exploitant ce modèle. Qui plus est, cette unification limite les duplications d'informations qui représentent une source classique d'erreurs dans les architec-



(a) Remote-Agent (b) IDEA

FIG. I.9 – Remote-Agent vs IDEA

tures hiérarchisées classiques. Par contre, IDEA semble présenter des difficultés de mise à l'échelle : la machine virtuelle, gérant le fonctionnement et les interactions des agents, risque fort d'être sujette à une explosion combinatoire sur un système complexe avec beaucoup d'agents. Par conséquent, il peut être nécessaire d'encapsuler l'ensemble des éléments fonctionnels dans un seul et même agent, comme illustré dans la figure I.9(b).

I.3 Problématique liée à la sûreté de fonctionnement

La sûreté de fonctionnement est une composante nécessaire de tout système informatique. Elle est définie comme la propriété permettant aux utilisateurs d'un système de placer une confiance justifiée dans le service qu'il leur délivre. C'est un concept générique qui est affiné dans [Avizienis 04] où les auteurs proposent l'arbre de la sûreté de fonctionnement (figure I.10). Cet arbre classifie les attributs, les menaces et les moyens d'améliorer la sûreté de fonctionnement d'un système.

Les attributs sont au nombre de 6 :

Disponibilité : *capacité à fournir un service correct dans un délai raisonnable.*

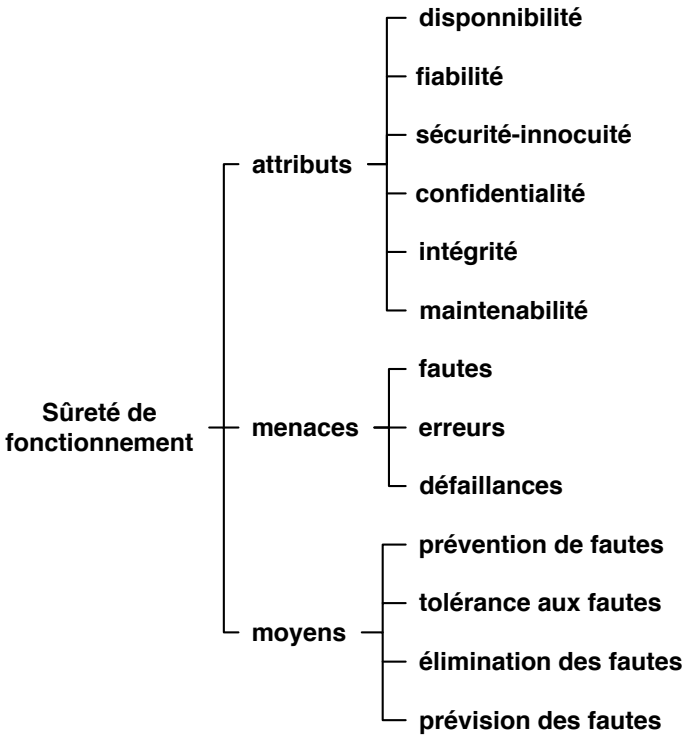


FIG. I.10 – L'arbre de la sûreté de fonctionnement

On entend ici par "raisonnable" que le temps d'attente doit être acceptable vis-à-vis des contraintes temporelles auxquelles le système est soumis. Ainsi, cet attribut devient critique dans un système embarqué temps réel où ce délai doit respecter des contraintes dures.

Dans les systèmes autonomes, on peut voir un lien entre la disponibilité et le besoin de réactivité. En effet, si un service n'est pas disponible à temps, le système ne pourra réagir correctement aux événements exogènes.

Fiabilité : *le service doit rester correct continûment au cours du temps.*

Ce concept met en évidence que, même si une situation adverse survient, le service doit toujours être assuré par le système. Il pourra dans le pire des cas être dégradé, mais le résultat final doit correspondre aux attentes de l'utilisateur.

Pour les systèmes autonomes, la fiabilité touche chaque composant – par exemple un module fonctionnel doit offrir certaines garanties sur sa fiabilité – ainsi que l'ensemble du système. Le travail présenté ici

s'intéresse particulièrement à cet attribut et nous développerons ici certaines des menaces à la fiabilité du système et la solution que nous proposons pour améliorer sa prise en compte.

Sécurité-innocuité : *absence de conséquence catastrophique vis-à-vis des utilisateurs et de l'environnement.*

Les conséquences catastrophiques peuvent être de plusieurs ordres. On y trouve entre autres des pertes et/ou dégradations physiques avec un coût financier ou humain inacceptable.

Les systèmes autonomes et, plus particulièrement, les robots sont particulièrement touchés par cet attribut. Si on prend par exemple un robot interagissant avec l'homme, il est nécessaire de garantir que ses actions ne risquent pas de blesser cette personne. Les capacités de décision de ce type de système entraînent que l'on se repose énormément sur le logiciel, et une faute dans la prise de décision mène à des situations qui menacent fortement la sécurité-innocuité. On peut considérer par exemple un robot qui, à cause d'une faute de conception, accélère à une vitesse de $2m.s^{-1}$. Si des personnes se trouvent dans son environnement direct, il peut les percuter et les blesser gravement.

Confidentialité : *absence d'accès non autorisé à des données privées.*

Les problèmes de confidentialité sont devenus populaires avec l'émergence des réseaux. Ils ne se limitent bien évidemment pas à ce domaine. Il est souhaitable que, si un système, accessible à un large public, contient des données dont l'accès est restreint à certaines personnes, les personnes non autorisées ne puissent pas avoir connaissance du contenu de ses informations. Il est nécessaire de protéger ses données en offrant des mécanismes limitant leur accès aux seules personnes concernées.

Cette problématique a été peu étudiée dans les systèmes autonomes jusqu'à présent. C'était principalement dû au fait que les plates-formes de ce type ne manipulaient pas ou peu de données de ce type. Toutefois, l'émergence de nouveaux centres d'intérêt comme le robot compagnon, l'interconnexion de systèmes autonomes via Internet ou les robots ubiquistes – qui exploitent des capteurs qui lui sont externes comme par exemple les caméras présentes dans le bâtiment – soulève ce type de problème.

Intégrité : *absence de changement non souhaitable du système.*

Ces changements non souhaitables sont des modifications du système

qui vont avoir pour conséquence que ce dernier ne fournit plus les services attendus. L'intégrité se trouve tant au niveau des données manipulées qu'au niveau du flot de contrôle. Un exemple simple sur l'intégrité est la mémoire des ordinateurs dont les bits peuvent changer de valeur à cause des radiations externes. Dans ce cas, les mémoires associent généralement des bits de CRC afin de se protéger de cette faute et ainsi assurer l'intégrité de la mémoire.

Dans les systèmes autonomes, cet aspect est pris en compte dans la majorité des architectures par les composants de diagnostic de faute comme le FDIR présent sur *Remote Agent*. Le rôle de ce système est de détecter une faute, de l'isoler et de proposer une reconfiguration du système afin que la fonctionnalité perdue soit retrouvée ou remplacée. L'intégrité du système est ainsi assurée et dans le cas d'une perte totale d'une fonctionnalité, le système peut le détecter et arrêter de proposer cette dernière jusqu'à la réparation éventuelle.

Maintenabilité : *possibilité de modifier et réparer le système.*

Cet attribut joue un rôle central lors du développement et la mise à jour des composants. Il est nécessaire d'offrir les outils permettant de faire évoluer le système aisément afin de corriger des problèmes éventuels ou ajouter de nouvelles fonctionnalités avec un risque faible de casser les fonctionnalités initiales.

Les architectures hiérarchisées ont toujours mis en avant cette problématique. Les trois architectures que nous avons présentées sont supportées par des outils logiciels ou formels facilitant la maintenabilité du système. Dans l'architecture LAAS, l'ensemble des outils proposés ont des bases formelles ou semi-formelles et sont bien intégrés les uns avec les autres. Par exemple l'interface de base entre les modules et le superviseur est généré automatiquement par l'outil GenoM. Dans CLARAty, l'exploitation de modèles objets pour la couche fonctionnelle va aussi dans ce sens. Finalement l'unification des modèles exploités par les agents IDEA permet de simplifier le travail de maintenance.

Chacun de ces attributs symbolise un idéal à atteindre. Les menaces à la sûreté de fonctionnement sont inévitables. Ces menaces sont reliées entre elles par la chaîne suivante : une faute va activer une erreur qui elle-même peut se propager déclenchant ainsi une défaillance du système qui peut activer une nouvelle faute. Pour casser ce cycle, le concepteur d'un système doit mettre en œuvre des moyens d'en limiter ses effets :

La prévention de fautes consiste à offrir des outils qui limitent les risques de

fautes. L'évolution des techniques de développement est fortement orientée par cet aspect. Elles offrent des bases de plus en plus solides qui permettent de limiter les fautes de conception. L'émergence des *garbage-collectors* permettant au développeur de se concentrer sur l'algorithme sans avoir à prendre en compte la gestion de la mémoire dynamique est un exemple de prévention.

La tolérance aux fautes consiste à intégrer dans le système des mécanismes lui permettant de corriger ou atténuer les effets d'une faute. Le diagnostic fait partie de cette catégorie. Une autre technique classique est le recouvrement qu'on peut retrouver avec des mécanismes de reprise d'erreur ou encore de redondance.

L'élimination des fautes consiste à identifier les fautes afin de les éliminer. La technique la plus classique est la vérification de logiciels. Elle peut s'appuyer sur des tests qui permettent de détecter une faute afin de la corriger. On trouve aussi les systèmes de diagnostic qui peuvent effectuer une reconfiguration du système afin qu'une faute soit éliminée.

La prévision des fautes consiste à donner une estimation des fautes et de leur impact sur le système. Pour ce faire, on va déterminer les sources possibles des fautes et, à partir des liens entre composants, déterminer comment celles-ci risquent de se propager dans le système. On trouve ici des techniques comme les arbres de fautes ou encore la méthodologie AMDEC[2].

La prise en compte de ces composantes de la sûreté de fonctionnement dans des systèmes à haut niveau d'autonomie est approfondie depuis peu par les chercheurs en robotique. Notamment, depuis 2001 le *workshop on robot dependability in human environment* (drhe) s'intéresse à cette thématique.

Les architectures hiérarchisées offrent généralement une bonne maintenabilité du système (couche fonctionnelle fortement modulaire, relative indépendance des composants, ...). Par contre, l'introduction d'aspects décisionnels de haut niveau pose toujours un problème vis-à-vis de la fiabilité et, en général, ce type de structuration, avec la duplication de données qui en découle, est peu adapté pour régler ce problème.

Dans cette section, nous présentons les problèmes liés à la fiabilité dans une architecture hiérarchisée classique. Ensuite, une présentation des divers travaux déjà effectués dans le domaine nous servira d'introduction à notre approche.

[2]AMDEC : Analyse des Modes de Défaillance, de leurs Effets et leurs Criticités.

I.3.1 Architecture hiérarchisée et fiabilité

Les architectures hiérarchisées embarquent des composants complexes qui interagissent généralement de façon asynchrone. La fiabilité de ce type de plateforme est souvent difficile à garantir. Pour illustrer cette lacune nous présentons ici les menaces à la sûreté de fonctionnement présentes généralement dans la couche fonctionnelle puis dans la couche décisionnelle. Cette analyse nous permettra d'offrir un premier bilan sur les lacunes de cette architecture vis-à-vis de la fiabilité du système.

Le niveau fonctionnel

Comme nous l'avons présenté en section I.2, ce niveau est décomposé en modules indépendants. Chacun d'entre eux offre un ensemble de services et – a priori – n'a pas de connaissance sur l'activité des autres modules. Ce choix correspond à un désir d'offrir une plus grande souplesse lors des phases de développement et d'intégration [Fleury 94]. En effet, grâce à cet aspect fortement distribué, il est aisé d'ajouter un module correspondant à une nouvelle fonctionnalité. Les seules modifications réelles se trouvent dans les composants décisionnels afin qu'ils exploitent ce dernier. Par contre, ce choix présente des menaces pour la fiabilité globale de la couche fonctionnelle :

– Un module n'ayant pas de connaissance de l'état actuel des autres modules, il n'a pas la possibilité de savoir si une de ses activités ne va pas générer de conflits avec celles d'un autre module.

– Les composants fonctionnels sont, par définition, amenés à interagir ou se synchroniser dans un contexte asynchrone. Le contrôle de ceci est généralement reporté vers un plus haut niveau où l'abstraction est pourtant plus grande. Ainsi, si une faute issue des composants décisionnels met le niveau décisionnel dans un état incohérent il n'y a quasiment aucun contrôle à ce niveau pour corriger ce problème.

Toutefois un module seul est un composant logiciel dont les diverses activités ont un comportement clairement défini [Mallet 02]. Ils s'appuient sur des modèles de fonctionnement avec un support formel qui permet d'offrir un certain contrôle sur ses activités. Dans l'architecture LAAS, par exemple, les services d'un module s'exécutent suivant un automate clairement défini et l'exécution concurrente d'activités au sein de celui-ci est gérée de façon déterministe. De plus, l'utilisateur peut spécifier des activités incompatibles à l'intérieur de ce module. Un système de contrôle interne au module assure qu'il n'y aura jamais deux activités incompatibles s'exécutant en même temps.

Un module seul est donc un composant logiciel relativement simple et clairement spécifié. On peut en extraire un modèle de comportement afin de valider son fonctionnement. L'état de l'art propose un grand nombre d'outils permettant ceci.

On peut prendre les outils de *model-cheking* symboliques permettant de confronter le modèle formel du système aux propriétés désirées du système. Les outils les plus connus de ce type sont SMV [McMillan 93], Spin [Holzman 97] et SAL [de Moura 04]. Chacun d'entre eux est puissant et permet la validation de propriétés exprimées en logique LTL ou CTL. Le langage SAL est le seul capable de manipuler des variables de taille non fixée ce qui le rend plus adapté à la validation de langages objets. La transition du code source du composant à valider vers le modèle synchrone est une tâche délicate à faire à la main, mais il existe plusieurs outils permettant de convertir un programme dans le format d'entrée d'un de ces outils. L'outil Bandera [Corbett 00] notamment traduit du code java en un automate à états finis directement exploitable. Cet outil permet aussi de retraduire les contre-exemples donnés par le *model-checker* afin qu'ils correspondent au code fourni. *Java PathFinder* [Havelund 00] effectue lui aussi une traduction du code java vers le formalisme exploité par SPIN. Ce model-checker est d'ailleurs exploité dans [Scherer 05] pour valider une boucle de contrôle d'un robot. Cette boucle semble toutefois relativement simple comparée aux systèmes que nous étudions ici. En effet, le système présenté dans cette article ne semble pas avoir de capacité décisionnelle avancée et la boucle de contrôle apparaît comme assez simple avec deux composants qui contrôlent la vitesse et l'orientation des roues en se basant sur des capteurs de lumière afin de suivre une ligne.

De même il existe un grand nombre d'outils de validation basés sur des modèles temporisés. Ce type d'outil permet d'ajouter la validation de propriétés portant sur les délais d'exécution. Cet aspect est central dans un contexte temps réel. Certains modules fonctionnels peuvent donc nécessiter ce type de validation. L'outil de validation le plus connu est UPPAAL [Bengtsson 95] qui permet la validation à partir d'une modélisation sous forme d'automates temporisés. AltaRica propose aussi une extension prenant en compte le temps [Pagetti 04]. L'outil TINA propose lui aussi ce type de validation [Berthomieu 03], il se distingue des autres par le choix des réseaux de Pétri temporels comme formalisme de modélisation.

Après ce tour d'horizon des divers outils de validation formelle, on peut affirmer sans problème qu'un module fonctionnel est validable. La situation se complique quand on se porte sur le contexte dans lequel il va s'exécuter. En effet, nous avons évoqué en début de section que, pour faciliter la modularité du système, un module n'a pas de connaissance explicite des autres composants fonctionnels tour-

nant sur le système. La connaissance de l'ensemble des services présents se trouve à un plus haut niveau. Il est donc difficile, au niveau fonctionnel, de connaître les effets que va avoir le lancement d'une activité même si celle-ci a été parfaitement validée de façon unitaire. Les activités tournant sur d'autres modules peuvent agir sur son comportement et générer une faute non prévue. Le tout s'exécutant de façon concurrente et généralement asynchrone, la validation de l'ensemble des composants de la couche fonctionnelle devient problématique. Ceci est d'autant plus vrai que, dans les systèmes autonomes évolués, les interactions entre modules sont complexes et mettent en œuvre un grand nombre de modules pour fournir un service donné. Le contrôle du tout est supervisé par la couche décisionnelle qui, comme nous allons le voir, présente de nouvelles menaces à la fiabilité globale du système.

Le niveau décisionnel

Le niveau décisionnel s'appuie sur des outils complexes exploitant des modèles de haut niveau. Cette abstraction est nécessaire à cause de la complexité des modèles et de l'espace de recherche exploré par ceux-ci. Elle présente toutefois une menace pour la fiabilité du système.

Ce niveau est généralement composé de trois types d'outils dont la tâche au sein de l'architecture est clairement définie. Nous présentons ici chacun d'entre eux en détail avec une analyse sur leur influence sur la fiabilité du système.

Le planificateur a un rôle primordial dans une architecture hiérarchisée. C'est ce composant qui va décider des actions à faire sur le long terme. La construction du plan est dirigée par les objectifs donnés au système.

La planification symbolique s'appuie sur une représentation abstraite du monde et des actions permettant d'agir sur celui-ci. La planification STRIPS [Fikes 72] se base sur l'hypothèse du monde clos[3]. Une tâche est symbolisée par ses préconditions, ses effets positifs et ses effets négatifs. Les préconditions indiquent l'état dans lequel doit être le monde pour que l'action soit réalisable. Par exemple, pour passer le pas d'une porte il faut que celle-ci soit ouverte. Les effets correspondent aux changements dans le monde que produit cette action. Un effet positif rend un fait vrai alors qu'un négatif supprime le fait concerné de la base de fait. A partir d'une représentation de l'état initial du monde, le planificateur choisit les actions à effectuer qui vont faire évoluer le monde jusqu'à obtenir l'état but qui satisfait les objectifs fixés. La modélisation a été depuis affinée pour se rapprocher au

[3]L'hypothèse du monde clos se base sur le principe que tout ce qui n'est pas vrai est faux. La représentation du monde se limite ainsi à ne spécifier que les prédicats vrais.

mieux de la réalité du problème. En particulier, le langage PDDL 2.1 [Long 03] s'est imposé comme une norme dans la communauté pour gérer de façon explicite le temps et les ressources.

Il existe de multiples techniques pour trouver le plan. Notre objectif n'est pas ici de faire un état de l'art poussé des techniques de planification, nous renvoyons le lecteur à un livre récent [Ghallab 04]. Nous retiendrons que, même si un problème simple peut être résolu avec une recherche en profondeur d'abord, les planificateurs évolués exploitent des techniques de recherche bien plus complexes permettant de produire des plans plus évolués. Le planificateur IxTeT, par exemple, explore l'espace des plans partiels [Trinquart 02]. La technique ici n'est plus dirigée par le but. Le planificateur va ici insérer des actions afin de gérer les incohérences dans le plan. Par exemple, si une variable X a une évolution qui est non encore expliquée, IxTeT recherche la ou les tâches permettant de justifier cette évolution. Le plan produit prend en compte l'évolution des ressources et les contraintes temporelles des actions. L'exécution parallèle d'action est prise en compte, ce qui permet d'avoir un résultat assez fin. Pour offrir une certaine souplesse à l'exécution, ce plan est partiellement instancié. Les variables qui n'ont pas besoin d'être contraintes ne le sont pas. Par exemple, si un robot qui a deux bras doit en utiliser un, le plan ne spécifie pas lequel et laisse ce choix à l'exécutif.

Les planificateurs modernes exploitent des techniques de plus en plus évoluées pour que le plan fourni corresponde au mieux à la réalité. Les modèles sont évolués, avec une représentation explicite des ressources et du temps. Des efforts sont aussi fournis pour que le résultat obtenu soit le moins contraint possible offrant ainsi une plus grande souplesse à l'exécution. Tout cela a un coût important. Les planificateurs modernes exploitent des CSPs et des algorithmes de recherche complexes. Il en résulte une complexité qui avoisine la NP-complétude. De même, la spécification du modèle devient de plus en plus complexe. Cette complexité n'est pas seulement due à la recherche d'un modèle réaliste du monde, l'utilisateur doit aussi prendre en compte les spécificités du planificateur afin que la recherche soit efficace. Le développement d'un modèle est donc une tâche complexe qui nécessite une certaine expertise.

L'abstraction du monde et des actions permet de simplifier ce modèle. Cette simplification est nécessaire autant par rapport à la complexité des algorithmes de recherche que pour simplifier la spécification du domaine. Par contre, il en résulte que le modèle du monde qu'a le planificateur colle peu au système. Seules quelques ressources sensibles sont représentées et le modèle des actions devient trop abstrait pour pouvoir prendre en compte les contraintes liées à l'exécution de bas niveau. Le développeur considère que la prise en compte de celles-ci sera assurée par les composants de plus bas niveau.

L'exécutif a pour rôle d'exécuter le plan fourni par le planificateur. Il affine les tâches de haut niveau en commandes de bases du niveau fonctionnel. Cette décomposition ne se limite généralement pas à un choix simple et déterministe. En effet, le besoin de robustesse pour les systèmes autonomes entraîne généralement des recouvrements fonctionnels qui offrent plusieurs moyens d'effectuer une tâche donnée. Chacun d'entre eux est plus ou moins adapté au contexte dans lequel se trouve le système et l'exécutif doit donc choisir le moyen le plus adapté selon ce qu'il perçoit de la situation courante. Pour y parvenir, il est souvent nécessaire d'utiliser des techniques avancées pouvant aller du simple mécanisme de reprise d'erreur jusqu'à des techniques d'apprentissage basées sur des modèles de Markov comme dans [Morisset 04].

La spécification des moyens d'exécuter une tâche du plan en actions de base est assez complexe. En effet, l'exécutif doit à partir d'une tâche de haut niveau unique, fournie dans le plan, affiner celle-ci en une séquence d'actions qui peut être totalement différente suivant le contexte. Unifier l'interface de modalités qui n'ont en commun que leur finalité représente un travail important. Prenons l'exemple d'une tâche de navigation. Celle-ci est donnée dans le plan par la simple directive "aller au lieu X". Par contre, au niveau de l'exécution, on va se retrouver avec des modalités d'actions très différentes suivant le type d'environnement dans lequel le système se trouve. Dans un environnement structuré à l'intérieur d'un bâtiment, une navigation basée sur une carte de ce dernier et un simple capteur de proximité pour détecter les obstacles éventuels peut être suffisant. En environnement extérieur accidenté, il sera par contre nécessaire de tenir compte du relief de l'environnement et, par exemple, d'exploiter un banc caméra stéréo afin de pouvoir modéliser celui-ci. Les modules fonctionnels et capteurs exploités sont donc totalement différents pour une tâche identique au niveau du plan. On ajoute que la modalité d'action que doit choisir l'exécutif peut changer durant une même tâche du plan (par exemple si le robot doit sortir du bâtiment durant cette tâche). Tous ces aspects mis ensembles font que l'exécutif, de par son rôle d'interface entre composants de très haut niveau et modules fonctionnels, est très difficile à développer. On doit à la fois tenir compte du niveau d'abstraction choisi dans le plan et de la multiplicité et diversité des interfaces qu'on trouve au niveau fonctionnel. La prise en compte de tous les cas d'exécution dans ces scripts est difficile et il est fréquent qu'une faute soit introduite durant la conception de ce composant. L'exécutif étant le composant qui commande le système cette faute présente un grand risque de propagation sur le système.

Le diagnostic est une autre composante importante du niveau décisionnel. L'objectif est ici de détecter les problèmes et, à partir d'un modèle du système, déduire

la cause réelle de celui-ci. Ainsi, les autres composants décisionnels peuvent exploiter cette information dans la reprise d'erreur. Son rôle se limite souvent à un contrôle de haut niveau qui est exploité par les composants décisionnels. Le modèle du système reste donc à un niveau d'abstraction assez élevé. La propagation de la faute se fait sur le plan en détectant les tâches qui sont menacées par la faute. A partir de celle-ci, il déduit comment corriger le problème (insertion d'une tâche de réparation, remise en cause de certains buts de faible priorité, ...) en fonction de la source du problème et de sa gravité.

Ce composant est très important vis-à-vis de la robustesse du système. En effet, c'est à travers lui que la couche décisionnelle prend en compte les problèmes lors de l'exécution du plan et déduit la façon de les régler au plus haut niveau du système. Le modèle du système permettant de détecter la source du problème est généralement difficile à produire [Bénazéra 03]. En effet, il faut prendre en compte l'arbre de fautes et les incertitudes sur les transitions qui ont été réellement activées. Ceci mène généralement à un modèle bayésien difficile à appréhender et où l'atteignabilité d'une situation donnée est un problème indécidable a priori.

Les interfaces entre ces divers éléments présentent une autre menace pour la fiabilité du système. En effet, chacun d'eux a sa propre représentation du système, avec un niveau d'abstraction plus où moins élevé, et va devoir interagir avec les autres. Ainsi, le travail à fournir pour la traduction d'un modèle vers un autre est important et fastidieux. L'ajout d'une nouvelle fonctionnalité nécessite la mise à jour de tous les modèles afin que celle-ci soit prise en compte correctement. Ce travail de traduction ne se limite pas seulement aux composants décisionnels, mais on le trouve aussi entre l'exécutif et les services de la couche fonctionnelle. Notre expérience sur la mise en œuvre de plates-formes basées sur ce type d'architecture nous montre que c'est ici qu'on trouve la plus grande menace vis-à-vis de la fiabilité globale du système. Dans [Bernard 00], les auteurs mettent en avant les difficultés et menaces que présente l'approche multi-modèle et la menace à la fiabilité qu'elle présente. Les auteurs y présentent un cas où l'ajout d'une nouvelle fonction au niveau du diagnostic a été mal intégrée dans l'exécutif. Cette extension a introduit une incohérence entre les modèles des deux composants qui fut difficile à détecter.

Premier bilan

Cette courte analyse montre que le niveau décisionnel et le niveau fonctionnel sont difficilement validables. A ceci s'ajoutent d'autres problèmes inhérents à la structure globale de ce type d'architecture :

– D'une part, ces deux couches sont appelées à interagir de manière asynchrone. Ainsi s'il est déjà difficile de valider chacune d'entre elles indépendamment, la tâche est encore plus complexe quand on doit tenir compte de cette interaction.

– D'autre part, la structuration en couche, ainsi que l'exploitation de différents paradigmes de programmation dans ce type d'architecture, nécessite la duplication des informations entre les couches. Ainsi, il est fréquent qu'une connaissance déjà présente dans la couche fonctionnelle soit dupliquée, sous une forme différente, dans la couche décisionnelle. Or on sait que ce type de duplication augmente les risques d'incohérence sur la donnée dupliquée et, par conséquent, menace la fiabilité globale.

Garantir la fiabilité d'une architecture hiérarchisée devient dès lors une tâche difficile. Il faut pouvoir assurer le bon fonctionnement de la couche fonctionnelle tout en assurant que les commandes envoyées à cette dernière par les composants décisionnels ne vont pas agir négativement sur sa fiabilité.

I.3.2 Architectures et fiabilité

La problématique liée à la fiabilité des systèmes autonomes n'est pas nouvelle et il existe déjà divers travaux sur le sujet. Dans un premier temps, nous présentons les approches systèmes, portant sur l'architecture en général, pour ensuite nous pencher sur des outils pouvant renforcer la fiabilité d'une architecture hiérarchisée.

Le point le plus limitatif quant à la fiabilité des architectures porte principalement sur la disparité des représentations des connaissances et la duplication des données qui en découle. Par conséquent, les travaux sur les architectures s'intéressent, en général, à offrir une formalisation de ceci afin de diminuer les risques de fautes liés à ce point lors des phases de conception et/ou d'intégration.

L'approche CLARAty

Dans CLARAty [Bualat 04], un effort tout particulier a porté sur l'exploitation des paradigmes objets dans l'architecture. En particulier, le niveau d'abstraction est donné dans la couche fonctionnelle par une hiérarchie objet. Ainsi, tous les composants offrant un même service héritent de la même interface abstraite.

L'apport immédiat vient du choix d'exploiter les concepts objets dans la couche fonctionnelle. Les paradigmes de programmation orientée objets offrent un cadre plus structuré lors de la conception du système. De nombreux outils offrent un bon support durant toute les phases du développement d'un projet basé sur des techniques objets. On peut citer le langage UML qui offre une représentation

semi-formelle durant la phase de conception afin que le système conçu corresponde aux spécifications initiales. Ce langage de spécification est aussi exploité par de nombreux outils – comme par exemple Rational ROSE-RT – permettant de générer le squelette initial du projet à partir des spécifications UML. Ainsi, la spécification de la couche fonctionnelle du système est supportée par des outils et techniques qui commencent à être bien connus par les développeurs et ont fait leurs preuves. L'apport, aussi faible soit-il, ne peut être que bénéfique sur l'ensemble des attributs de la sûreté de fonctionnement et, plus particulièrement, sur la fiabilité du système conçu avec ces outils.

Un autre intérêt de cette architecture est lié à la représentation explicite du niveau d'abstraction. Ceci permet une meilleure interaction entre niveau décisionnel et fonctionnel. En effet, le parallèle entre abstraction de tâches et la hiérarchie de classes, permet d'offrir la possibilité au plan décisionnel d'accéder à des informations directement dans le plan fonctionnel. Cette possibilité est à comparer avec les architectures hiérarchisées classiques où, pour qu'une donnée soit accessible par le planificateur, elle doit d'abord transiter par l'exécutif. Dans CLARAty on évite cet intermédiaire inutile et l'on diminue ainsi les risques qu'une faute altère celle-ci.

Enfin, les efforts récents pour proposer une librairie générique de composants fonctionnels [Bualat 04] présente aussi un grand intérêt vis-à-vis de la fiabilité. Cette librairie offrirait une structuration de base – comparable aux motifs de conception[4] – qui permettrait ainsi d'avoir un cadre précis pour la mise en place des diverses fonctionnalités du système. L'exemple sur lequel les auteurs travaillent actuellement est d'offrir une hiérarchie générique pour le composant qui contrôle la locomotion d'un robot à roues. Ce travail est déjà fastidieux car il est nécessaire d'offrir une interface de haut niveau assez générique pour s'abstraire du nombre de roues mais aussi de propriétés qui vont jouer un rôle plus important sur la commandabilité du robot (par exemple un robot holonome sera plus simple à manœuvrer qu'un système dont les commandes sont comparables à celles d'une voiture). Toutefois, s'ils parviennent à proposer une hiérarchie d'abstraction assez générique pour ceci, les concepteurs de nouveaux systèmes pourront s'appuyer sur cette dernière et ainsi garder une généricité à haut niveau avec un effort relativement faible.

On peut constater, que dans CLARAty, la mise en évidence de la hiérarchie d'abstraction permet d'offrir de bonnes bases pour la phase de conception. L'ensemble des efforts liés à l'amélioration de la fiabilité semblent d'ailleurs portés

[4]Un motif de conception, ou design pattern, est un concept issu de la programmation objet destiné à résoudre les problèmes récurrents. A la différence des algorithmes, un design pattern propose plutôt des procédés de conception généraux et éprouvés.

sur cet aspect. C'est une composante nécessaire car une bonne conception permet de diminuer les risques de fautes dans le système ou de propagation de cette dernière. Par contre, à notre connaissance, cette architecture ne présente pas de composant offrant un contrôle actif permettant d'avoir une tolérance aux fautes durant l'exécution.

L'approche IDEA

IDEA est née des conclusions tirées de la mise en place de *Remote Agent* pour DS1 [Bernard 00]. Si on regarde les conclusions de cet article on peut clairement voir les idées générales exploitées dans IDEA :

- Une première leçon fut le constat que la validation unitaire des composants n'est pas suffisante. Il faut donc proposer un ensemble d'outils permettant de fournir des tests adéquats ainsi qu'une validation formelle de l'ensemble. Pour ce faire les auteurs indiquent que les interfaces doivent être clairement spécifiées.
- La validation des différents modèles exploités dans *Remote Agent* fut particulièrement délicate. La conséquence fut que certaines modifications, considérées comme mineures, n'ont pas été testées. Certaines ont pourtant eu un impact négatif sur la fiabilité du système. Les auteurs indiquent donc clairement qu'il est nécessaire d'offrir des outils facilitant la validation des divers modèles ainsi que le comportement de l'architecture dans sa globalité.
- Spécifier les modèles de haut niveau exploités par le planificateur demande des connaissances spécifiques sur le formalisme utilisé et sur le domaine en temps que tel. On se trouve face à une situation où la personne qui connaît le domaine n'est pas la même que celle qui est capable de le développer. Les auteurs proposent donc de choisir un formalisme de représentation plus simple et compréhensible. Ils avancent d'ailleurs le choix des *timelines* qui ont une représentation relativement intuitive.

Le choix d'un modèle unique, formel et simple comme les *timelines* est la solution que propose IDEA à ces différents problèmes. En effet, tous les agents ayant une même représentation, leurs interactions sont plus simples à formaliser et la validation du système dans sa globalité en est facilitée. De plus les outils et techniques utilisés pour valider un agent fonctionneront dans d'autres contexte. Finalement, les *timelines* permettent d'avoir une représentation des divers modèles simple et accessible.

Les interdépendances entre les différentes actions sont symbolisées par des relations de Allen[5]. Si on revient sur la figure I.8 (page 23), on voit que ces

[5]Les relations de Allen sont des opérateurs binaires permettant de poser des contraintes élémentaires entre deux intervalles. Les 13 relations primitives sont *e(qual) ; b(efore) ; m(eets) ;*

relations sont représentées par des arcs entre les diverses *timelines*. Ainsi, on voit dans ce plan qu'une stéréo corrélation (*scorrel*) est nécessairement directement précédée par une prise d'images (*shot*).

Cette unification des modèles permet aussi de n'utiliser qu'un seul mécanisme de fonctionnement pour les divers composants. La seule différence se trouve dans l'horizon sur lequel l'agent est capable de planifier et l'heuristique choisie pour la recherche de solution. Ainsi, les agents de haut niveau vont planifier sur le long terme alors que les agents de bas niveau vont plutôt être des composants réactifs. Ce moteur de fonctionnement unique permet d'avoir un meilleur contrôle sur l'ensemble du système et offre une plus grande cohérence globale.

IDEA propose une solution innovante aux problèmes de fiabilité présents dans les architectures hiérarchisées. Le modèle choisi dans cette architecture semble présenter des bases formelles suffisantes pour permettre d'en automatiser la validation. Le système exhibe donc une autonomie de haut niveau tout en offrant la possibilité de garantir sa fiabilité. Par contre, on ne trouve pas non plus de contrôle en ligne permettant de limiter la propagation d'une faute dans le système.

L'approche LAAS

L'architecture LAAS met particulièrement l'accent sur la mise en place de mécanisme de contrôles en ligne.

On trouve déjà cet aspect au niveau de l'outil GenoM. Lors de la spécification des divers services, l'utilisateur peut expliciter que deux services sont incompatibles au sein d'un même module. Lors de la demande de lancement d'un service, le module interrompt toute activité incompatible avec ce dernier pour permettre son lancement. Ce contrôle reste toutefois limité au sein du module et ne s'étend pas à l'ensemble des interactions entre les divers composants du niveau fonctionnel.

Pour assurer la fiabilité du système, l'architecture LAAS propose d'intégrer un système de contrôle à l'interface entre la couche décisionnelle et la couche fonctionnelle. Le rôle de se composant est d'assurer la sécurité-innocuité des modules fonctionnels afin que la fiabilité de ce niveau soit améliorée. Le placement de ce contrôleur entre les deux couches est basé sur le constat que les composants fonctionnels sont commandés par l'exécutif et que la majorité des fautes lors de l'exécution sont dues à une mauvaise représentation des interdépendances entre service au niveau décisionnel. Le contrôleur intervient alors comme un filtre

o(verlaps) ; *s(tarts)* ; *d(uring)* ; *f(inishes)* ; et leur inverse *bi* ; *mi* ; *oi* ; *si* ; *di* ; *fi*.

sur les commandes de la couche décisionnelle permettant d'assurer qu'aucune de celles-ci ne menace la fiabilité du système.

Un premier contrôleur, basé sur KHEOPS [Ghallab 88], a été proposé dans [de Medeiros 96]. KHEOPS est un outil permettant de compiler un ensemble de règles de productions (condition(s) ⇒ action(s)) en un latticiel[6]. Une telle structure permet de garantir que l'inférence du système s'effectue en un temps maximum borné (qui est du même ordre que la profondeur maximale du graphe). Grâce à cette propriété, le développeur peut garantir que le contrôleur résultant respecte les contraintes temps réel du système et, par conséquent, évaluer les possibles pertes en réactivité de ce dernier.

La spécification des contraintes du niveau fonctionnel était donnée par un graphe d'incompatibilité entre services. La figure I.11, donne un exemple de ce graphe. Il y a ici deux modules qui offrent chacun 3 fonctions de bases. L'ensemble du niveau fonctionnel exhibe 10 services (de A à J). Les arcs orientés représentent les incompatibilités entre fonctions et/ou services. Ce graphe est ensuite traduit en règles de productions compilées par KHEOPS.

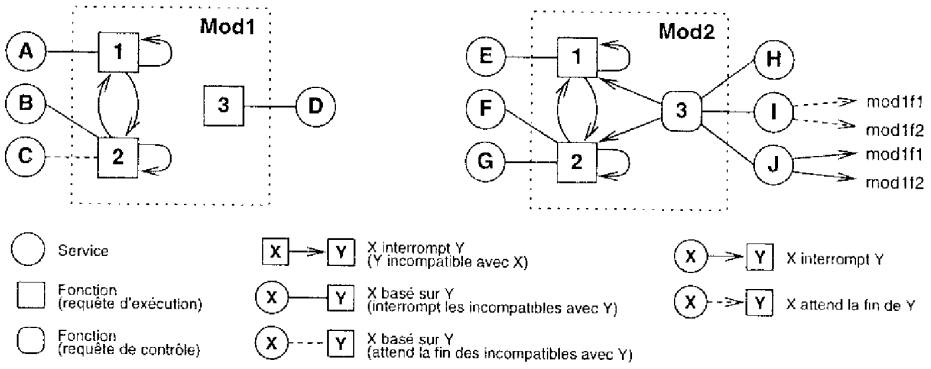


FIG. I.11 – Exemple de graphe d'incompatibilité

On peut constater ici que le seul type d'interdépendance géré par ce contrôleur est l'incompatibilité entre activités. Ce contrôle permet certes d'éviter les problèmes liés aux accès concurrents à une ressource non partageable mais il reste trop limité pour gérer toutes les interdépendances possibles entre services. Il est par exemple impossible de spécifier une règle telle que "le service X ne peut être exécuté si on n'a jamais lancé le service Y". Pour pallier à ce problème l'auteur propose de reporter les contraintes temporelles au niveau de l'exécutif.

[6]Un latticiel est un graphe orienté acyclique.

Cette idée se base sur le fait que le langage de spécification de PRS peut être réduit de telle sorte qu'il est modélisable sous forme de réseaux de Pétri. Il propose donc de modéliser les scripts de bas niveau en réseaux de Pétri colorés afin de pouvoir valider ceux-ci. Nous pourrions ainsi garantir que les commandes envoyées par la couche décisionnelle respectent bien les contraintes temporelles de cette couche.

L'approche proposée a été peu exploitée au sein même de l'architecture LAAS. On peut en donner plusieurs causes :

– Le contrôleur choisi exhibe des capacités de contrôle relativement faibles. En effet, on ne considère ici que les services ne pouvant s'exécuter en même temps. Ceci est en parti lié au fait que le langage KHEOPS ne permet pas de spécifier directement des contraintes temporelles comme la précédence.

– La mise en place de ce système est relativement complexe. D'une part le graphe de conflit devait être spécifié directement en règles KHEOPS. Or les développeurs de modules fonctionnels, qui sont le plus à même de donner ces contraintes, ne sont pas familiers avec les règles de production.

– De même la modélisation des procédures PRS en réseau de Pétri colorés est loin d'être simple et demande un travail assez lourd. L'auteur propose des représentations génériques de divers opérateurs de base mais celles-ci, bien que limitées à la version la plus simple de chaque opérateur, sont déjà très complexes.

L'architecture LAAS propose de façon explicite un composant assurant le contrôle du fonctionnement de la couche fonctionnelle. Ce contrôle permet de garantir que les commandes lancées par les composants décisionnels ne vont pas créer un conflit. C'est, à notre connaissance, la seule architecture à proposer ce type de composant. Nous nous appuyons ici sur cette structuration et proposons d'étendre les possibilités de contrôle en ligne à ce niveau.

I.3.3 Techniques et approches pour la fiabilité

Comme nous venons de le voir, dans l'architecture LAAS la fiabilité s'appuie sur l'exploitation d'un composant tiers qui maintient le système dans un état sûr en ligne. Plusieurs techniques ont été proposées afin d'avoir un contrôle de ce type sur le fonctionnement du système. Beaucoup d'entre elles s'appuient sur un modèle synchrone afin de bénéficier des hypothèses simplificatrices de cette approche.

Nous présentons ici différents outils et techniques qui permettent d'améliorer la fiabilité du système durant son exécution. Les outils permettant la validation

hors ligne – tels que UPPAAL ou encore SMV – ont déjà été présentés brièvement dans la sous-section I.3.1. Nous ne les développerons pas plus ici car, même si la validation des composants est nécessaire pour améliorer sa validité, ce type d'approche est relativement éloignée que celle présentée dans ce manuscrit où l'objectif est d'offrir un contrôle en ligne.

ORCCAD

ORCCAD [Espiau 95] s'intéresse tout particulièrement à l'exploitation de langages synchrones pour le développement de plates-formes robotiques. Cette architecture propose un environnement de développement, basé sur Esterel [Boussinot 91], permettant de spécifier aisément les composants fonctionnels du système.

Pour ce faire, le langage Esterel a été étendu avec divers concepts adaptés à la conception de robots :

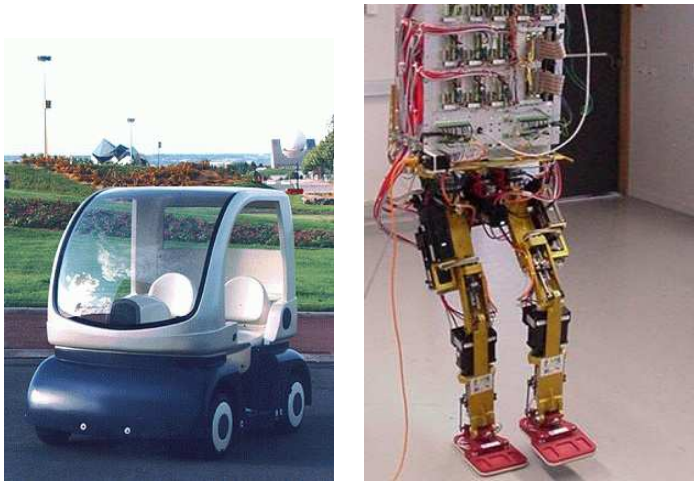
– les "tâches" du robot (RT) modélisent les actions de base où on trouve les boucles de contrôle de la plate-forme. Une RT spécifie le comportement de ces boucles ainsi que l'ensemble des événements correspondant à ses préconditions, postconditions et exceptions. Ceux-ci donnent une représentation événementielle de la RT.

– les "procédures" du robot (RP) décrivent les actions plus complexes du système. Une RP est une composition de RTs. Au plus haut niveau celle-ci représentera une mission du robot qui abstrait les liens logiques et temporels des RTs sous-jacentes.

Ces deux nouvelles structures permettent une représentation incrémentale du système. De plus, l'environnement s'appuyant sur un langage synchrone, l'exécution concurrente des diverses tâches est simplifiée par la composition synchrone et il devient plus facile de valider formellement l'ensemble du système.

ORCCAD a d'ailleurs été utilisé avec succès sur de nombreux robots. On peut donner entre autre le véhicule robotisé CyCab dans le projet LARA ou encore le robot BIP qui est un robot marcheur avec 15 degrés de liberté, tous deux illustrés à la figure I.12. Chacun de ces robots effectue des tâches complexes comme la navigation autonome ou encore la marche, et l'environnement ORCCAD permet d'offrir de bonnes garanties sur leur bon fonctionnement.

Même si cet outil est exploité sur des systèmes robotiques complexes. On peut constater que ORCCAD est plutôt centré sur l'aspect fonctionnel du système. Ainsi la prise en compte des possibles menaces issues de composants décisionnels n'est pas gérée par ce composant. Ceci doit être lié au fait qu'à notre connais-



(a) CyCab (b) BIP

FIG. I.12 – Robots basés sur ORCADD

sance, il n'existe pas d'instance de cette architecture embarquant des composants décisionnels avancés.

La synthèse de contrôleur discret

L'hypothèse synchrone est aussi exploitée dans les travaux présentés dans [Altisen 03]. Ici, l'objectif est d'exploiter les techniques de synthèse de contrôleur discret [Marchand 00] dans un système robotique.

La synthèse de contrôleur discret permet de générer le composant qui va empêcher le système d'atteindre un état invalide. Pour ce faire, on a un modèle discret du système ainsi que les propriétés qu'on souhaite maintenir sur ce dernier. Les auteurs exploitent l'environnement MATOU [Rémond 01] pour la modélisation du système et SIGALI [Marchand 00] pour la génération du contrôleur.

MATOU est un outil basé sur le formalisme des automates de mode. Il permet de traduire un automate de mode dans le langage de programmation synchrone Lustre. Le formalisme des automates de mode est un langage synchrone basé sur les flots de données qui explicite les modes de comportement. La figure I.13 donne un exemple simple. On voit deux modes `incr` et `decr`, à chaque mode est associé un comportement : dans `incr` x est incrémenté à chaque cycle et dans `decr` x est décrémenté. Les transitions sont étiquetées par les conditions de passage. Ainsi sur cet exemple, x, qui vaut initialement -1, est incrémenté jusqu'à 10 puis décrémenté jusqu'à 0 et ainsi de suite.

Ce langage permet de modéliser facilement des systèmes dont le comportement

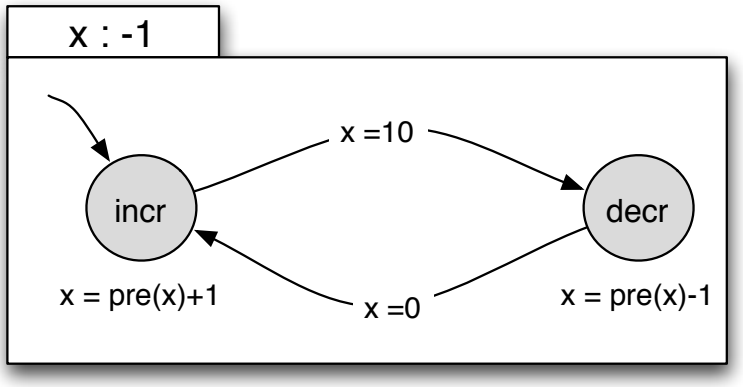


FIG. I.13 – Exemple simple d'automate de mode

change à l'occurrence d'un ou plusieurs événements. Il supporte la composition synchrone qui consiste à considérer que si deux événements se produisent au même instant, ils ne sont pas considérés de façon séquentielle mais comme "vraiment" simultanés et leurs effets sont immédiats. Par exemple, dans la figure I.14, la mise en parallèle des automates $\{A, B\}$ et $\{C, D\}$ est équivalente à l'automate unique $\{AC, BD\}$ car l'événement b, produit par $\{A, B\}$, est immédiatement consommé par $\{C, D\}$.

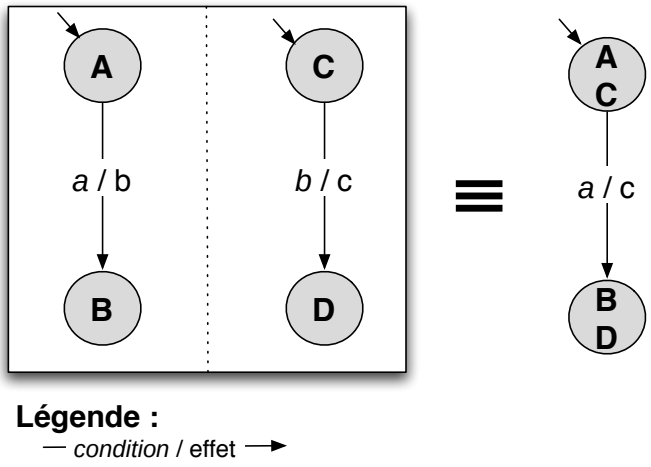


FIG. I.14 – La composition synchrone

L'outil SIGALI est un *model checker* symbolique qui permet la synthèse de contrôleur discret. Le contrôleur généré est le plus permissif au sens où il satisfait les contraintes données du système tout en conservant le maximum de compor-

tements du système original. Le principe est de générer à partir de SIGALI un composant logiciel qui agit sur les événements dont il a le contrôle pour maintenir le système contrôlé dans un état satisfaisant les propriétés désirées. Ainsi, la composition du système initial et du contrôleur devient un nouveau système dont la fiabilité a été augmentée. Nous reviendrons plus en détail sur cette technique dans la section II.2.1.

Ces travaux démontrent clairement que de telles techniques sont exploitables sur des systèmes aussi complexes que les robots. Toutefois, ce contrôleur se connecte à l'interface de la couche fonctionnelle sans forcément prendre en compte la couche décisionnelle. En effet – comme nous le verrons particulièrement au chapitre IV (p. 85) – l'interaction entre un contrôleur d'exécution et les composants décisionnels n'est pas triviale et ne peut être comparable à celle entre ce même contrôleur et un opérateur humain.

Un exécutif basé sur les modèles

Une autre approche intéressante est présentée dans [Williams 03b]. Ici, l'approche se base plutôt sur l'exploitation des modèles exploités dans le système de diagnostic pour exécuter les demandes en limitant les risques de fautes. L'exécutif de bas niveau qu'ils exploitent est illustré par la figure I.15. L'exécutif titan est

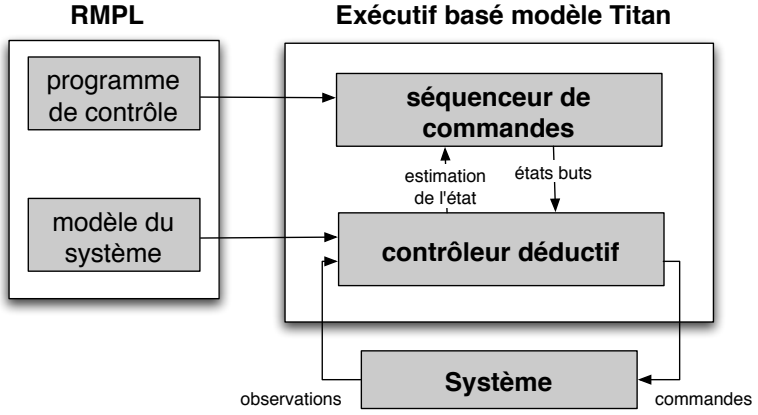


FIG. I.15 – L'exécutif Titan

composé de deux éléments :

– *le contrôleur déductif* est le composant qui est connecté aux composants physiques. Il exploite le modèle de Markov caché afin de déduire l'état probable du système. De même il est capable de planifier de façon réactive les actions

```
ComputeCorrection():: {
  do {
    Correction = Computed
  } watching Correction = Failed;
  if Correction = Computed thennext CorrectionStatus = Succeeded
    elsenext CorrectionStatus = Failed
}
```

FIG. I.16 – Exemple de code RMPL

à effectuer pour atteindre un état but. Pour ce faire, les auteurs ont spécifié l'heuristique *conflict-directed* A^* [Williams 03a] qui permet de rechercher de façon réactive le chemin dans l'espace d'états qui approche du but en limitant les risques de problèmes.

– *le séquenceur de commandes* exécute les programmes RMPL [Williams 03a] et fournit les nouveaux buts au contrôleur en fonction de la dernière estimation de l'état du système. Le langage RMPL permet de spécifier des objectifs en fonction de l'état du système pour effectuer une tâche en se basant fortement sur le modèle sous-jacent. Ce langage est très proche d'ESTEREL[Boussinot 91]. La différence se trouve sur l'orientation état de RMPL quand ESTEREL est plutôt basé sur les événements. De plus, la syntaxe de RMPL interdit les programmes non déterministes ou incohérents comme **present** `S` **else** `S` **end** en ESTEREL. Dans ce langage la forme équivalente est **if** `S` **thennext** `S` où le retrait de la simultanéité du test et de l'action élimine ce risque. Le compilateur a ainsi un travail de vérification moins complexe à effectuer. La figure I.16 donne un exemple simple de fonction RMPL pour corriger la trajectoire de DS-1. Le but est donné par `Correction=Computed`. On peut constater que ce dernier est encadré par l'opérateur `do { } watching` qui permet d'abandonner cet objectif initial si le contrôleur indique un échec.

Le contrôleur Titan exploite efficacement les techniques issues du diagnostic classique pour permettre une exécution fiable des actions. Ce système permet d'offrir un contrôle de l'exécution des actions basé sur une spécification des composants qui explicite les probabilités de pannes de ceux-ci. Tout ceci se fait en plus avec une prise en compte de l'observabilité partielle de l'état réel du système. Par contre cette approche se limite surtout à la modélisation du matériel et ne semble pas forcément adaptée à la prise en compte de l'interaction des composants logiciels du système à un plus haut niveau.

Observer l'exécution

Dans [Bensalem 05], les auteurs présentent une approche qui permet plutôt la validation du fonctionnement du système. Leur composant observe l'exécution du plan afin de valider ce dernier.

Pour ce faire, le plan produit est traduit en un automate temporisé [7] [Alur 94] qui va être mis en confrontation avec les retours donnés par l'exécution de ce dernier. Ces automates sont générés de façon automatique à partir de la spécification du plan. Les auteurs se basent ici sur le *K9 Rover*. Il considèrent que le plan produit est un HTN (Hierarchical Task Network). Dans ce formalisme le plan est un nœud d'un arbre où un nœud correspond soit à une tâche élémentaire soit à un bloc composé d'une séquence de nœuds. Chaque nœud spécifie un ensemble de conditions d'exécution (préconditions, invariants, ...). Chacune de ces conditions explicite les contraintes temporelles associées. Les auteurs proposent sur cette base une méthode permettant de compiler ces plans en automates temporisés pour chacun des nœuds de l'arbre.

A partir de ces automates temporisés, on génère un composant dont le rôle sera d'observer l'exécution du plan. Cette génération s'appuie sur la technique d'estimation d'état proposée dans [Tripakis 02]. Cette technique consiste à déterminer l'ensemble des états qui correspondent à une observation donnée. Si cet ensemble est non vide alors il est possible d'obtenir ces observations, dans le cas contraire elles sont impossibles au vu du modèle. Cette technique permet de générer un observateur en considérant les contraintes soit de façon analogique (où le temps est vu comme continu), soit de façon discrète (le temps est décomposé en ticks). Les auteurs présentent comment générer ces deux types d'observateurs à partir des automates temporisés.

Le composant ainsi généré permet ainsi d'avoir une trace de l'exécution du plan. Cette trace indique les nœuds du plan exécutés et si ils ont réussis ou échoués avec les dates des différents événements associés à chacun d'eux (start, success, failure, ...). L'analyse de ces traces permet ensuite de vérifier si celles-ci correspondent aux spécifications du système. Ainsi, lors de leurs expérimentations, les auteurs ont pu constater que l'exécution d'une tâche ne respectait pas ses spécifications. En effet celle-ci s'est terminée aussi vite qu'elle avait commencée alors que la spécification indiquait qu'elle durait au moins 1 seconde.

On se trouve ici plutôt avec un outil permettant de garantir la validité du plan par rapport à son exécution. On peut ainsi affiner les modèles des actions utilisés

[7] Un automate temporisé se compose de 2 éléments : un automate fini classique – décrivant les états et transitions du système – et des horloges permettant de spécifier les contraintes temporelles associées aux transitions de façon quantitative

en fonction des diagnostics donnés par l'observateur. Leur composant n'intervient pas dans l'exécution et se limite à fournir un bilan sur la bonne exécution du plan fourni. Par contre on trouve une gestion poussée des contraintes temporelles grâce à une traduction automatique du plan en automate temporisé.

I.4 Présentation de notre approche

Notre travail se situe clairement dans le contexte de l'architecture LAAS. Nous proposons ici d'implémenter un mécanisme contrôlant l'interaction entre la couche décisionnelle et fonctionnelle afin d'éviter que le système n'atteigne un état indésirable. Le principe directeur de cette approche s'appuie sur une idée simple :

> Si on ne peut valider le couple fonctionnel/décisionnel, autant mettre en place un composant simple qui aura connaissance de façon synchrone et en temps réel de tout changement possible d'état du système et empêchera d'atteindre des états spécifiés comme invalides, dangereux ou indésirables.

Pour que ce composant soit exploitable correctement il doit respecter les propriétés suivantes :

Observabilité : A tout moment, il doit pouvoir observer les événements susceptibles de changer l'état du système.

Contrôle : Il doit pouvoir changer l'état du système afin d'éviter que ce dernier entre dans un état indésirable.

Temps borné : Ce composant ne doit pas être une entrave à la réactivité du système. Qui plus est, si on souhaite exploiter un modèle synchrone, le système doit traiter un changement d'état dans un temps négligeable. En effet, si ce temps est trop long l'état du système peut évoluer de telle manière que le résultat obtenu n'est plus pertinent.

Validable : Ce contrôleur agit directement sur l'état du système en vue d'en augmenter la fiabilité. Il devient dès lors nécessaire de pouvoir vérifier son comportement afin de garantir que ce dernier soit correct. De plus, si les propriétés précédentes sont respectées, une telle validation garantira le comportement de l'ensemble contrôleur/composants fonctionnels.

Simplicité : L'outil supportant le développement de ce composant doit être relativement aisé à utiliser, dans le cas contraire il ne sera pas exploité car trop complexe à mettre en œuvre.

Intégration : Il est nécessaire que ce composant s'intègre bien dans l'ensemble de l'architecture. Ceci ne se limite pas à la prise en compte des composants fonctionnels, sur lesquels il agit, mais doit s'étendre à la prise en compte des composants décisionnels. En effet, même si le contrôleur ne peut agir directement que sur l'état des composants fonctionnels, il faut garder à l'esprit que ces actions vont nécessairement interférer avec les actions des composants de plus haut niveau. Ainsi notre contrôleur doit intégrer cet aspect afin que ses actions ne soient pas une entrave à la prise de décision.

Pour atteindre ces objectifs, la synthèse de contrôleur discret présente des propriétés intéressantes. Si on peut s'appuyer sur des hypothèses synchrones, l'exploitation des techniques de model-checking symbolique [Shnoebelen 99] permettent de générer notre contrôleur en s'appuyant sur des bases formelles tant au niveau de la spécification du comportement du système contrôlé qu'à celle des contraintes de ce dernier. Le résultat s'appuyant sur un système de preuve, un contrôleur ainsi généré offre la garantie que celui-ci respectera bien les propriétés désirées.

A notre connaissance, aucun des travaux portant sur le contrôle synchrone ne prend en compte l'interaction entre les composants fonctionnels – qui agissent sur le système contrôlé – et le contrôleur. Dans notre cas cette problématique devient importante. Nous verrons plus tard que les actions du contrôleur vont à l'encontre des décisions prises au plus haut niveau. Nous verrons plus précisément dans la chapitre IV que ceci peut avoir des effets néfastes sur le fonctionnement du système. Il faut donc prendre en compte cet aspect dans ce type d'architecture.

Chapitre II

Contrôler l'exécution

Nous présentons dans ce chapitre les hypothèses que nous avons sur le système. Celles-ci serviront de base pour nos travaux. Nous pourrons ensuite présenter le contrôleur qu'on se propose d'intégrer dans cette architecture en indiquant les éléments qui le compose ainsi que son principe de fonctionnement.

II.1 Hypothèses sur l'architecture

L'approche présentée ici a été développée sur l'architecture LAAS (cf. page 18). Par conséquent, les hypothèses que nous développons dans cette section s'appuient fortement sur le fonctionnement de cette architecture. La couche de contrôle d'exécution étant le cœur de notre travail, les présupposés que nous présentons ici portent sur les deux autres niveaux du système. Toutefois, même si ceux-ci se basent principalement sur les outils développés au LAAS, ils restent assez généraux pour être adaptés à une autre architecture hiérarchisée.

Les deux niveaux gérant le fonctionnement de base du système sont le fonctionnel et le décisionnel. Nous commençons donc par présenter les hypothèses faites sur le fonctionnement de chacun d'entre eux qui nous permettront ensuite de proposer un système assurant la fiabilité de l'ensemble durant son exécution.

II.1.1 Le niveau fonctionnel

Nos hypothèses s'appuient sur le fonctionnement de l'outil GenoM utilisé pour générer les modules fonctionnels dans notre architecture [Fleury 96]. La couche fonctionnelle est décomposée en modules, chacun d'entre eux regroupant l'ensemble de services associés à une fonctionnalité ou un composant physique du système. On trouve par exemple un module gérant les caméras alors qu'un autre permettra la planification de chemins pour la navigation du robot.

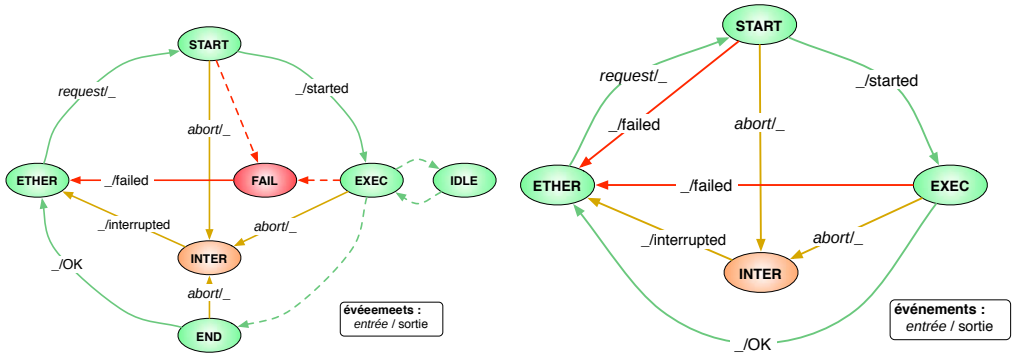
Les modules de la couche fonctionnelle sont indépendants les uns des autres. Chacun d'entre eux offre un ensemble de services pouvant être lancés via une *requête* où sont donnés les arguments nécessaires à son lancement. A la fin de son exécution l'activité correspondante renvoie un *bilan* de son exécution accompagné d'informations correspondant au type de retour ; pour une terminaison nominale ce sera des informations propres au traitement demandé alors qu'en cas d'échec, on trouvera plutôt un code d'erreur permettant d'identifier la cause de ce dernier. Les types et domaines de ces données sont clairement définis lors de la spécification du module pour chacun des services. Les activités s'exécutent de manière concurrente sur le système.

Afin de garantir l'indépendance des modules, ils n'ont pas de connaissance a priori des autres modules présents sur le système. Celle-ci est localisée dans les composants décisionnels comme le superviseur. Nous posons donc comme hypothèse qu'une requête est toujours issue de l'extérieur de la couche fonctionnelle et ne peut en aucun cas venir d'un autre service fonctionnel. Par contre, il peut être nécessaire à l'exécution d'un service qu'il exploite des données produites par un module tiers (par exemple la stéréo corrélation a besoin des images prises par les caméras). Afin de permettre ceci, les modules exportent des données publiques en lecture appelées *posters*. La connexion entre un service et un poster se fera dynamiquement via les arguments de la requête associée.

Un autre point à considérer ici est l'observabilité et la contrôlabilité sur l'état des services. En effet, on ne peut contrôler quelque chose qu'on ne peut observer ou sur lequel on ne peut agir. Prenons par exemple l'automate représentant une activité GenoM donné à la figure II.1(a). Son fonctionnement peut être résumé de la façon suivante :

– Initialement on se trouve dans l'état ETHER qui indique une absence d'activité pour le service.

– A la réception d'une requête (*request*), le contrôleur du module amène l'activité dans l'état START. C'est ici que sont faits les contrôles initiaux sur la possibilité de lancer l'activité.



(a) Modèle interne (b) Vue de l'extérieur

FIG. II.1 – Modèle d'une activité

– Si les contrôles initiaux n'ont pas posé de problèmes, un message indiquant le bon lancement de l'activité est envoyé (**started**) et on bascule dans l'état EXEC. C'est ici que les traitements correspondants à l'exécution réelle du service sont effectués. Le contrôleur d'activité positionnera régulièrement ce processus dans l'état interne IDLE afin de permettre aux autres activités du module de s'exécuter.

– A la fin nominale de l'activité, on bascule dans l'état END où les données associées au bilan de succès sont fixées et envoyées avec le bilan final **OK**. Le service retourne ensuite dans l'état ETHER.

– Une activité peut être interrompue à tout moment. Ceci peut être dû à une demande explicite du client (*abort*) ou au contrôle interne du module. En effet, il est possible lors de la spécification du module d'indiquer que deux services sont incompatible entre eux. Quand un service est lancé toute activité incompatible avec lui est immédiatement interrompue. Dans ce cas, on se trouve dans l'état INTER où les ressources réservées par le système sont libérées avant l'envoi d'un bilan final indiquant son interruption (**interrupted**).

– Une autre fin non nominale peut survenir lorsqu'un problème a été détecté lors de l'état INIT ou EXEC qui empêche la bonne exécution du service. Dans ce cas on bascule dans l'état FAIL qui va libérer les ressources et indiquer les causes du problème envoyées avec le bilan d'échec (**failed**).

On peut constater ici que certaines transitions sont gérées en interne par le contrôleur d'activité du module et ne sont ni contrôlables ni observables directement de l'extérieur. Ainsi après avoir reçu le message **started**, il est impossible de savoir si l'activité est dans l'état EXEC, IDLE, FAIL ou END. Par contre, on peut réduire cet automate afin de n'obtenir que des transitions observables et/ou

contrôlables de l'extérieur. On obtient l'automate de la figure II.1(b) qui exprime mieux une activité telle qu'elle est perçue par les clients du module.

Notre composant effectuant un contrôle externe sur les services, on se limite donc à cette dernière représentation du comportement d'une activité. On peut ainsi avoir une connaissance à tout moment de son état ainsi que des moyens qu'on a d'agir dessus. On constate d'ailleurs que les moyens d'actions sur un service sont relativement limités. Les deux seuls événements sur lesquels on a un contrôle sont `request` et `abort`.

II.1.2 Le niveau décisionnel

Les hypothèses présentées ici sont moins critiques que celles portant sur le niveau fonctionnel. Il est toutefois nécessaire de prendre en compte la présence des composants décisionnels. Ces hypothèses sur les composants décisionnels serviront de base aux réflexions sur l'interaction entre le niveau décisionnel et notre contrôleur présentées dans le chapitre IV.

Comme pour le niveau fonctionnel, nous nous appuyons ici sur l'existant dans l'architecture LAAS. Ainsi nos réflexions se basent sur une instance réelle d'architecture. Dans la figure II.2, on voit que le niveau décisionnel contient trois principaux composants [Lemai 04] :

– *Le planificateur* est le centre de la prise de décision de haut niveau. C'est lui qui, à partir d'une représentation du monde et des moyens qu'il a pour agir sur ce dernier, va rechercher la séquence d'actions permettant d'atteindre les objectifs qui lui ont été fournis en entrée. De tels outils peuvent prendre en compte des contraintes complexes autant sur des aspects temporels que sur les ressources [Laborie 96]. Leur espace de recherche est vaste et complexe. Ceci a des conséquences directes sur la complexité algorithmique généralement compensée par une abstraction forte dans la représentation du monde.

– *L'exécutif temporel* contrôle la cohérence temporelle du plan durant son exécution. C'est lui qui lance les tâches à la date voulue et vérifie que l'état réel du système correspond à celui donné par le plan. En cas d'incohérence, qu'elle soit due à un dépassement de délai, à la modification des buts ou à l'apparition d'une situation non anticipé, ce composant va demander au planificateur, selon le niveau de criticité de la menace, de réparer le plan ou d'en chercher un autre. Il peut aussi décider d'abandonner certains buts de faible importance si ceux-ci contraignent trop le problème. On a ainsi une solution aux problèmes liés à l'exécution d'un plan.

– *Le superviseur* gère le lien entre les composants de haut niveau et la couche fonctionnelle. Il reçoit des tâches de haut niveau qu'il va affiner en une séquence

d'actions directement exécutables. Il est capable de faire une reprise d'erreur en cherchant un autre moyen d'effectuer la tâche qui lui a été demandée. Il a aussi pour rôle de traduire les données et bilans issus de la couche fonctionnelle en informations nécessaires à la mise à jour la représentation du monde de l'exécutif temporel. Comme dans [Morisset 04], ce composant peut exploiter des algorithmes d'apprentissage afin de sélectionner la modalité la plus appropriée selon le contexte.

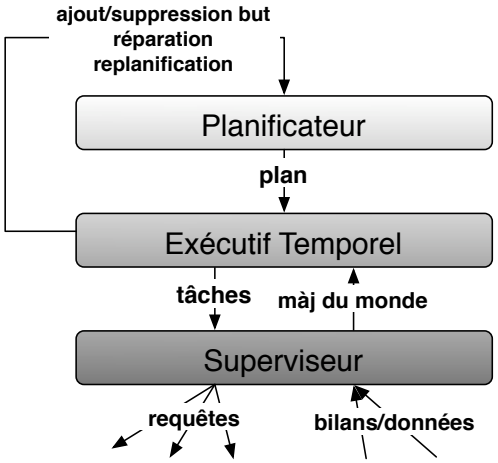


FIG. II.2 – Niveau décisionnel

L'hypothèse majeure que nous posons ici est que la connaissance des intentions reste localisée dans ce niveau. En effet seul le planificateur a une connaissance précise des objectifs à long terme du système. De même, le lien entre une action de base et la tâche associée à un plan est géré par le superviseur. Hors de ce niveau l'information sur les intentions du système n'est plus présente et on ne voit que les actions de bases sans connaître leur finalité.

II.2 Un contrôleur comme filtre d'exécution

Nous avons vu dans la section précédente que le lancement d'activités dans le niveau fonctionnel est directement lié aux décisions prises dans la couche décisionnelle. La principale menace à la fiabilité de la couche fonctionnelle vient d'ailleurs de la faible connaissance qu'ont les composants de haut niveau sur les interactions et conflits qu'on peut trouver entre les activités de la couche fonctionnelle. Cette lacune a deux principales causes :

– d'une part, la complexité des composants fonctionnels, tant au niveau de la spécification des domaines qu'à celui de l'espace de recherche, pousse les développeurs à s'abstraire du fonctionnement interne afin de privilégier la représentation du monde et des effets des actions. Par conséquent, on se retrouve avec des décisions prises sans prendre en compte les possibles menaces que celles-ci présentent quant à la fiabilité du système.

– d'autre part, il y a rarement une réelle unification entre les interfaces des modules fonctionnels. Ceci rend la modélisation de ceux-ci et de leurs interactions d'autant plus difficile. Même si des efforts sont effectués dans ce sens dans des architectures comme CLARAty [Bualat 04], nous avons déjà indiqué que cette unification des interfaces entre services similaires est difficile. Qui plus est, une telle modélisation n'est pas suffisante pour assurer la fiabilité durant l'exécution. Il est nécessaire d'avoir en plus un outil permettant de garantir que le système n'atteindra pas un état non nominal.

Nous proposons ici de mettre en place un système qui va contrôler le fonctionnement de la couche fonctionnelle afin d'empêcher cette dernière d'atteindre un état indésirable.

II.2.1 Filtrer l'exécution

Contrôler l'exécution revient à mettre en place un composant qui va filtrer les événements du système afin d'éviter que ce dernier n'atteigne un état indésirable. Si on considère :

– un système S qui pourra être modélisé, par exemple, par un automate résultant de la composition synchrone de l'ensemble de ses activités.

– un ensemble de propriétés P correspondant au comportement désiré de S. Ces propriétés spécifient par exemple les états qui ne doivent jamais être atteints par le système.

Le contrôleur C est le composant qui garantit que les propriétés P seront respectées par $S \times C$, la composition de ce dernier avec S. Une technique classique pour générer le contrôleur C est d'exploiter les résultats de la validation de S vis-à-vis de P afin de connaître les situations où ces propriétés deviennent invalides. Ces résultats serviront de base pour le contrôleur. Pour ce faire on doit avoir une bonne connaissance sur les moyens qu'a le contrôleur d'agir sur S. C agira donc sur les contrôlables de S afin que ce dernier respecte les propriétés P.

On peut illustrer ceci par la figure II.3 où **S** représente l'ensemble des états atteignables par S et **P** l'ensemble des états respectant P. Notre objectif est de créer le système $S \times C$ tel que l'ensemble des états atteignables par ce dernier –

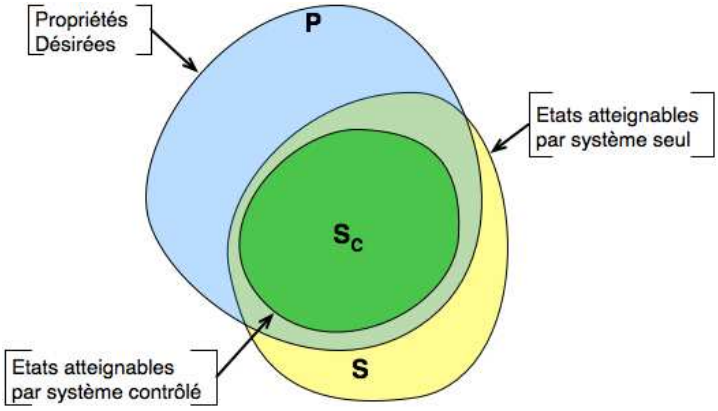


FIG. II.3 – Recherche d'un contrôleur

appelé $\mathbf{S_C}$ – respecte la propriété suivante :

$$\mathbf{S_C} \subseteq \mathbf{P} \cap \mathbf{S}$$

Le contrôleur qui modifiera le moins le comportement du système sera tel que :

$$\mathbf{S_C} \equiv \mathbf{P} \cap \mathbf{S}$$

On a ainsi un contrôleur tel que $S \times C$ permet d'atteindre tous les états valides de S. Toutefois un tel contrôleur est généralement impossible à obtenir. En effet, il existe des états valides de S qu'on ne peut atteindre sans transiter par des états invalides ou encore, certains états valides qui peuvent mener à un état invalide via une séquence d'événements non contrôlables. Lors de la synthèse de contrôleur on cherche tout de même à maximiser $\mathbf{S_C}$ afin que $S \cap C$ ait le comportement le plus proche possible de S quand celui-ci ne menace pas P. Le contrôleur le plus permissif [Marchand 00] correspond à cette définition. On peut le voir comme un contrôleur qui n'agira sur l'évolution de S que si celle-ci menace les propriétés P.

La synthèse de contrôleur est spécialement adaptée quand on est face à des systèmes dits réactifs. Un système réactif va lancer des traitements simples activés par des événements afin de produire les sorties attendues. De tels systèmes peuvent être modélisés en s'appuyant sur les hypothèses synchrones. On considère ainsi que le système s'exécute sur une machine telle que la vitesse de traitement est négligeable vis-à-vis de la fréquence du tic horloge. Si on peut respecter ces hypothèses, le modèle de notre système est une machine à états finis. A partir de là on peut exploiter des techniques comme le *model-checking* symbolique afin de déterminer le comportement du contrôleur.

On distingue plusieurs types de propriétés que l'on peut spécifier pour un système donné [Shnoebelen 99] :

Sûreté : dans un contexte donné, le système n'atteindra jamais une situation indésirable.

Vivacité : dans une situation particulière, quelque chose finira par avoir lieu.

Absence de blocage : le système ne se trouvera jamais dans un état à partir duquel il ne peut plus progresser. Cette propriété peut, dans la majorité des cas, être transformée en propriété de sûreté. Il suffira d'indiquer que l'état menant à un blocage est indésirable.

Équité : sous certaines conditions, une situation aura lieu (respectivement n'aura pas lieu) une infinité de fois. Ce type de propriété peut être vu comme une vivacité infinie.

La synthèse de contrôleur se limite souvent à assurer uniquement les propriétés de sûreté. Ceci est principalement dû à la difficulté d'assurer les propriétés de vivacité sur un système qu'on ne contrôle que partiellement. Qui plus est, si elles ne sont pas bornées, les propriétés de vivacités n'assurent que le fait que la situation arrivera "un jour" ce qui, dans une application réelle, n'est pas suffisant pour assurer le bon fonctionnement de celle-ci.

II.2.2 Intégration dans une architecture hiérarchisée

Nous présentons ici le composant que nous avons développé et intégré dans l'architecture LAAS afin d'expérimenter et valider nos travaux. Ce composant, appelé le *Request & Report Checker* (R^2C) et illustré par la figure II.4, est constitué des éléments suivants :

Le buffer d'entrée a pour rôle de capturer les événements susceptibles de changer l'état du système tel qu'il est perçu par le R^2C. Ces événements sont :
- les requêtes issues de la couche décisionnelle,
- les bilans intermédiaires et finals des activités tournant dans la couche fonctionnelle
- ainsi que les changements de valeurs de posters lus par une activité.

Le rôle de ce buffer est de capturer et stocker les événements survenant entre 2 cycles du R^2C.

La représentation de l'état du système est la base de connaissances du R^2C. On y trouve toutes les informations sur l'état courant du système nécessaire au contrôle de l'état. Pour optimiser le contrôle, ces données

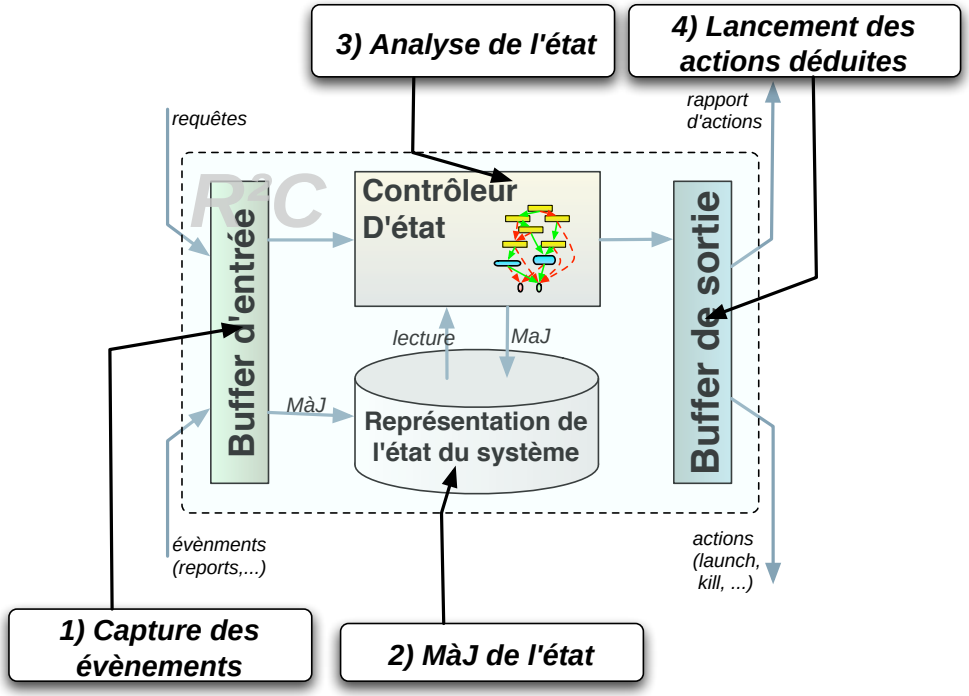


FIG. II.4 – Le R^2C

sont traitées afin que leur lecture par le contrôleur d'état se déroule en un temps constant. Nous développerons cet aspect plus tard dans cette section.

Le contrôleur d'état stocke les règles qui régissent le bon fonctionnement du système. Ces règles sont représentées sous une forme compacte qui offre des garanties sur le temps maximum de traitement. C'est ce composant qui analyse l'état du système prévu et déduit les actions à effectuer pour ne pas menacer les règles qu'il maintient.

Le buffer de sortie a pour rôle de redistribuer les événements et messages du R^2C aux composants du système. On y trouve :

- les bilans intermédiaires et finals capturés initialement par le buffer d'entrée,
- les requêtes lancées par la couche décisionnelle et qui ne menacent pas les règles gérées par le contrôleur d'état,
- les événements déduits par le contrôleur d'état,
- et les bilans des actions du R^2C informant les composants décisionnels des actions que le contrôleur d'état a déduit qui vont à l'encontre

du comportement attendu (rejet d'une requête, interruption d'une activité, ...).

Le fonctionnement de l'ensemble de ces éléments se base sur le cycle suivant :

1. *Capture des événements :* ceux-ci correspondent aux demandes de lancement de services, aux bilans d'activités et à tout autre changement susceptible de modifier l'état du système (modification d'un poster, ...).
2. *Mise à jour de l'état du système :* afin de déduire le futur état du système, le R^2C intègre les événements. La représentation de ce dernier est limitée au minimum nécessaire pour assurer le contrôle. Ainsi un événement qui ne menace pas ces propriétés ne modifie pas cette représentation et sera juste transféré vers son destinataire réel sans traitement particulier.
3. *Analyse de l'état :* si l'état a changé, le R^2C vérifie que ce dernier est valide. Dans le cas contraire, il agit sur les événements dont il a le contrôle afin de maintenir le système dans un état consistant avec les propriétés. Par exemple, si le superviseur envoie une requête sur le service X incompatible avec le service Y en cours d'exécution, le contrôleur peut agir soit en rejetant X soit en interrompant l'activité Y.
4. *Lancement des actions :* Le contrôleur lance les actions déduites. On y trouve les événements capturés qui ne menacent pas les propriétés à maintenir ainsi que les événements qu'il a produits afin de maintenir ce dernier dans un état consistant. On trouve aussi des messages supplémentaires indiquant à la couche décisionnelle les actions effectuées.

On peut différencier ces composants en deux catégories. D'un côté nous avons les éléments synchrones purs qui sont la base de connaissance et le contrôleur d'état. Ceux-ci se basent sur l'hypothèse synchrone où on considère que tout événement survient durant un tic horloge et que notre temps de traitement est inférieur à l'intervalle entre 2 tics. Ainsi les données lues dans la représentation de l'état du système ne changeront pas durant tout le temps où le contrôleur d'état s'exécute. Ceci nous permet de représenter les règles sous une forme telle qu'une variable d'état ne sera testée qu'une fois. D'un autre côté les deux buffers servent d'interface vers le système réel. Le buffer d'entrée en particulier stocke tous les événements qui surviennent avant l'occurrence du prochain tic. Même si les événements n'arrivent pas en même temps dans ce buffer, ils seront perçus de façon synchrone à l'intérieur du R^2C.

Toutefois il faut pouvoir garantir qu'aucun événement reçu par le buffer d'entrée alors que le R^2C est en train de vérifier la validité d'un état ne remette en cause ce dernier. Dans ce but, nous considérons ici que le temps de traitement du R^2C est négligeable par rapport à la fréquence des événements. On peut relaxer

cette hypothèse en se basant sur le principe même des architectures hiérarchisées. En effet, une architecture hiérarchisée est structurée de telle sorte que plus on monte dans la hiérarchie, moins les contraintes temps réels sont importantes. Ainsi on peut considérer que les événements issus de la couche décisionnelle sont moins urgents que ceux issus de la couche fonctionnelle plus temps réel. Cette vision est renforcée par le fait que le R^2C ne contrôle que la couche fonctionnelle. Ainsi quand on reçoit un événement de cette couche celui-ci correspond à une évolution réelle de l'état du système contrôlé, alors qu'un événement issu de la couche décisionnelle et capturé par le R^2C n'a pas encore affecté l'état réel du système. On peut par conséquent considérer qu'un événement issu de la couche décisionnelle alors que le R^2C est en train de traiter un état ne remet pas en cause la décision qui sera prise. Il nous suffit donc de garantir que le temps de traitement est inférieur au délai minimum entre deux événements issus uniquement de la couche fonctionnelle.

Représentation de l'information dans le R^2C

Nous nous intéressons ici plus précisément au contrôleur d'état et à la représentation de l'état de la couche fonctionnelle. Ces deux composants représentent le cœur de notre approche et stockent toutes les informations spécifiques. Nous allons voir ici plus en détail leur fonctionnement ainsi que leur représentation internes.

Le contrôleur d'état s'appuie fortement sur la représentation de l'état de la couche fonctionnelle. En effet, si ce dernier stocke et manipule les règles de fiabilité de la plate-forme, les informations sur les états courant et futur, sont situées dans la représentation de l'état du système. Par conséquent, pour une plus grande efficacité il faut que cette représentation corresponde bien aux attentes du contrôleur d'état.

Nous verrons dans la section III.2.2 (page 77) que le contrôleur d'état s'appuie sur les OCRDs, une structure de donnée arborescente équivalente aux OBDDs en ajoutant le support de contraintes numériques fixées. Les OCRDs décomposent ces contraintes pour générer une partition du domaine de valeurs de la variable. Ceci nous permet de différencier les services suivant les arguments qui ont été passés à la requête. Cette structure de donnée nous donne les informations de base pour construire le contrôleur d'état et la représentation de l'état du système :

– la partition de l'espace des valeurs donnée par l'OCRD nous donne la structure finale de notre base de données représentant l'état.

– l'arbre de décision de l'OCRD porte l'information des contraintes de fiabilité du système et sera directement exploité par le contrôleur d'état.

La façon dont l'OCRD est généré sera décrite en détail dans le chapitre suivant. Ce que nous devons en retenir est que cette structure est un arbre binaire où chaque nœud correspond à un test sur un élément de la partition. La base de données offre autant d'entrée que d'élément de cette partition. A chaque entrée est associé l'ensemble des activités qui correspondent à celle-ci. L'entrée est une valeur booléenne qui est vraie si et seulement si l'ensemble associé est non-vide.

La mise à jour de ces ensembles est effectuée par un mécanisme de *callback*. A chaque type d'événement est associé une fonction qui détermine l'élément de la partition auquel ce dernier est associé. Si un événement change l'état courant d'une activité, celle-ci est retirée de l'ancien ensemble auquel elle était associée et la fonction indique le nouvel ensemble auquel elle appartient. La mise à jour de cette base de données s'effectue donc en dynamique à partir de la partition de l'espace d'état déduite lors de la génération du contrôleur.

Dans le chapitre suivant nous allons voir le modèle sur lequel nous nous sommes appuyés pour représenter l'état de la couche fonctionnelle dans l'architecture LAAS. Nous présenterons ensuite comment nous allons générer et représenter les règles ainsi que la partition de l'espace d'état lors de la génération de notre contrôleur.

Chapitre III

Spécification & génération du contrôleur

Notre contrôleur d'exécution est spécifique à une instance de système donnée. Les contraintes du système dépendent des modules et des composants physiques présents sur la plate-forme. Par exemple, un même module qui agit sur la vitesse de déplacement d'un robot peut être exploité sur un robot pouvant aller à de grandes vitesses, mais aussi sur un autre où une vitesse supérieure à un seuil donné aura des conséquences dramatiques. Par conséquent, dans le second robot, il faudra ajouter une contrainte pour que cette vitesse ne soit pas dépassée.

Il faut un moyen de spécifier les contraintes que le R^2C devra maintenir sur la plate-forme destination. Dans cette partie nous présentons avant tout le modèle qu'exploite notre contrôleur pour représenter le fonctionnement du système. A partir de ce modèle, on donne les outils formels ainsi que le langage exploité pour générer le contrôleur filtrant l'exécution du système. Finalement nous présentons comment le R^2C exploite la structure de donnée produite par notre outil afin de maintenir le système dans un état consistant sans que les performances globales de notre système en pâtissent.

III.1 Modèle initial

Avant de pouvoir spécifier les contraintes du système il est nécessaire d'avoir un modèle de son fonctionnement. En effet ce modèle permet au contrôleur d'avoir une représentation du fonctionnement de la couche fonctionnelle, il peut dès lors déduire comment il peut interagir avec cette dernière afin de la maintenir dans un état nominal. Dans cette section nous présentons donc un modèle de la couche fonctionnelle décrite dans la section II.1.1. Dans un but de clarté la définition de ce modèle se découpe en deux volets :

1. un premier s'attache à définir le fonctionnement des activités/processus liés aux services.
2. le second s'appuie sur ce modèle assez bas niveau pour d'écrire les états de chaque service.

Ce dernier modèle nous servira de base afin de spécifier les contraintes du système que devra maintenir le contrôleur.

III.1.1 Modèle de fonctionnement de la couche fonctionnelle

Nous présentons ici un modèle formel simple représentant le comportement de la couche fonctionnelle en s'intéressant plus particulièrement aux activités en cours d'exécution ainsi qu'au passé de celles-ci.

Pour représenter ceci nous définissons le système $\mathbf{S} = \{\mathbf{T}, \mathbf{B}, \mathbf{R}, \mathbf{P}, \mathbf{A_{arg}}, \mathbf{A_{ret}}\}$ avec :

$\mathbf{T} = \mathbb{N}^+$	L'ensemble des instants.Chaque instant est étiquetée par une valeur entière suivant son occurence.
$\mathbf{B} = \{\top, \bot\}$	L'ensemble des valeurs booléennes.
$\mathbf{R}$	L'ensemble des services de la couche fonctionnelle.
$\mathbf{P}$	L'ensemble des activités (ou processus).
$\mathbf{A_{arg}} = \bigcup_{r \in \mathbf{R}} \mathrm{D}^{\mathrm{r}}_{\mathrm{arg}}$	L'ensemble des arguments des requêtes. $\mathrm{D}^{\mathrm{r}}_{\mathrm{arg}}$ définit le domaine de validité de l'argument passé à la requête pour le service r
$\mathbf{A_{ret}} = \big(\bigcup_{r \in \mathbf{R}} \mathrm{D}^{\mathrm{r}}_{\mathrm{ret}}\big) \cup \{\mathrm{FAIL}\}$	L'ensemble des valeurs de retour des requêtes. $\mathrm{D}^{\mathrm{r}}_{\mathrm{ret}}$ définit le domaine de validité de la valeur de retour nominale du service r

On définit la fonction *instancesof* : $\mathbf{R} \times \mathbf{A_{arg}} \rightarrow 2^{\mathbf{P}}$. Cette fonction donne l'ensemble des activités instances d'un service avec un argument donné sur toute la durée de vie du système. En utilisant cette fonction, on définit les effets d'une demande de service avec $req : \mathbf{R} \times \mathbf{A_{arg}} \times \mathbf{T} \rightarrow \{\top, \bot\}$. Ce prédicat indique si une demande de service a été lancée à une date donnée. Son effet sur le système est donné par la formule suivante :

$$\forall (r, a, t) \in \mathbf{R} \times \mathbf{A_{arg}} \times \mathbf{T} : \\ req(r, a)_t \Leftrightarrow \big(\exists p \in instancesof(r, a) : \neg active(p)_t \wedge active(p)_{t+1}\big) \quad \text{(III.1)}$$

On définit de même le prédicat $end : \mathbf{P} \times \mathbf{A_{ret}} \times \mathbf{T} \rightarrow \{\top, \bot\}$. Ce dernier indique l'occurrence de la fin d'une activité et respecte les propriétés suivantes :

$$\forall (p, t) \in \mathbf{P} \times \mathbf{T} : \\ \big(\exists ret \in \mathbf{A_{ret}} : end(p, ret)_{t+1}\big) \Leftrightarrow \big(active(p)_t \wedge \neg active(p)_{t+1}\big) \quad \text{(III.2)}$$

$$\forall (x, y) \in \mathbf{A^2_{ret}} : \big(end(p, x)_t \wedge end(p, y)_t\big) \Rightarrow x = y \quad \text{(III.3)}$$

Avec ces trois règles on a une représentation de comment une activité démarre (III.1) et se termine (III.2). On assure aussi qu'il y a une et une seule valeur de retour à la fin de celle-ci (III.3). Afin de garantir qu'une activité correspond à une exécution unique et continue d'un service on ajoute la règle suivante :

$$\forall p \in \mathbf{P}, \forall (t, t', t'') \in \mathbf{T^3}, t < t' < t'' : \\ \big(active(p)_t \wedge active(p)_{t''}\big) \Rightarrow active(p)_{t'} \quad \text{(III.4)}$$

Dans la section II.1, nous avons indiqué que pour tout service, il existe un service spécifique permettant d'interrompre une activité attachée à ce premier. Nous représentons ceci dans notre modèle via la fonction *killer* : $\mathbf{R} \rightarrow \mathbf{R}$ qui donne le service permettant de tuer les instances du service passé en argument. Le service $killer(r)$ recoit comme argument l'activité que l'on souhaite interrompre. Ainsi le domaine des arguments de ce type de service contient l'ensemble des activités associées à r. On peut exprimer ceci ainsi :

$$\forall (r, k) \in \mathbf{R^2} : \big(killer(r) = k\big) \Rightarrow \Big(\bigcup_{x \in D^r_{arg}} instancesof(r, x)\Big) \subseteq D^k_{arg} \quad \text{(III.5)}$$

On constate dans cette équation que l'ensemble des activités r est inclus dans D^k_{arg}. Ceci est dû au fait qu'un service d'interruption (*killer*) peut interrompre plusieurs types de services. Par exemple dans GenoM chaque module a un seul

service de ce type permettant d'interrompre tous les types de services associés à ce module.

L'effet d'une requête d'interruption sur l'instance qui lui est passée en argument est spécifié comme suit[1] :

$$\forall (r,x) \in \mathbf{R} \times \mathbf{A_{arg}}, \forall p \in \mathit{instancesof}(r,x) : \\ \mathit{active}(p)_t \wedge \mathit{req}(\mathit{killer}(r),p)_t \Rightarrow \left(\exists \mathit{ret} \in D^r_{ret} \cup \{\text{FAIL}\} : \mathit{end}(p,\mathit{ret})_{t+1}\right) \tag{III.6}$$

Cette règle illustre le fait que le lancement d'une demande de service d'interruption va provoquer la fin de l'activité passée en argument. On peut noter que la valeur de retour de l'activité interrompue est indéterminée. Ceci est dû à la possibilité que la date du lancement de la requête d'interruption corresponde à la fin nominale de l'activité passée en argument. Dans tous les autres cas, la valeur de retour sera nécessairement FAIL qui correspond à un bilan d'échec.

Les règles (III.1) et (III.6) sont centrales pour notre contrôleur. En effet c'est en jouant sur ces deux règles – plus particulièrement sur le rejet des requêtes (req) ou le lancement de requêtes d'interruption ($killer$) – que ce dernier peut contrôler le système. Il nous reste maintenant à avoir une meilleure vision du passé du système. Nous introduisons pour cela la fonction

$$\mathit{lastended} : \mathbf{R} \times \mathbf{T} \rightarrow (\mathbf{A_{arg}} \times \mathbf{A_{ret}}) \cup \{\emptyset\}$$

Elle se comporte comme une mémoire stockant les informations sur la dernière instance correctement exécutée pour un service donné. Elle respecte le comportement suivant :

$$\forall (r,t) \in \mathbf{R} \times \mathbf{T} : \\ \mathit{lastended}(r)_t = \begin{cases} \emptyset & \text{si } \nexists (p,x,y,t') \in \mathbf{P} \times D^r_{arg} \times D^r_{ret} \times \mathbf{T}, t' \leq t : \\ & \quad p \in \mathit{instancesof}(r,y) \wedge \mathit{end}(p,y)_{t'} \\ \emptyset & \text{si } \exists (p,x) \in \mathbf{P} \times D^r_{arg} : \\ & \quad p \in \mathit{instancesof}(r,x) \wedge \mathit{active}(p)_t \\ (\mathit{arg},\mathit{ret}) & \text{si } \exists (p,\mathit{arg},\mathit{ret}) \in \mathbf{P} \times D^r_{arg} \times D^r_{ret} : \\ & \quad p \in \mathit{instancesof}(r,\mathit{arg}) \wedge \mathit{end}(p,\mathit{ret})_t \\ \mathit{lastended}(r)_{t-1} & \text{sinon.} \end{cases} \tag{III.7}$$

[1] On considère qu'une activité en cours d'exécution peut être tuée dans tous les cas

La valeur de retour $\emptyset$ indique que le passé est indéterminable. C'est le cas lorsque aucune activité ne s'est terminée jusqu'à présent ou si une activité est en cours d'exécution. Ce choix est lié au fait que, lorsqu'une activité s'exécute, le passé du service auquel elle correspond peut être remis en cause à tout moment par la fin de cette dernière.

Dans la section II.1.1, nous avons vu que les activités communiquent entre elles via des flots de données. Ces flots unidirectionnels sont appelés *posters* dans l'architecture LAAS. L'accès à ces *posters* se fait via un identifiant généralement passé en argument à la requête qui lance l'activité cliente. Afin de pouvoir exprimer ceci nous nous limitons à la modélisation des clients sans expliciter le comportement du producteur. En effet, une erreur dans un poster est issue du producteur, par contre les conséquences de cette erreur se répercutent sur le client. Il est donc plus important d'avoir une connaissance précise des clients. Qui plus est, lors de la spécification des contraintes, l'utilisateur pourra tout à fait expliciter le lien qui existe entre un *poster* et son producteur (par exemple en indiquant qu'une activité cliente donnée ne peut s'exécuter que si le producteur de ce poster est actif). Pour pouvoir accéder à la valeur d'un *poster*, on définit la fonction *posterval* qui prend en argument un identifiant et retourne la valeur associée à l'instant présent. Cet opérateur va pouvoir être utilisé dans le modèle donné dans la section suivante.

III.1.2 Spécification des contraintes

Le modèle que nous avons présenté précédemment colle bien au comportement du système et donne de bonnes indications quant aux moyens d'influer sur son état. Toutefois ce modèle reste de trop bas niveau pour permettre de spécifier aisément les contraintes liées aux possibles interactions et conflits présents. Pour palier à ceci, nous allons nous appuyer sur ce modèle pour en spécifier un nouveau plus général. Ce modèle s'intéresse plus aux requêtes qu'aux activités sous-jacentes. De plus, afin d'apporter cette abstraction sur les valeurs des arguments et retours de services, on remplace dans $\mathbf{S}$ les domaines $\mathbf{A_{arg}}$ et $\mathbf{A_{ret}}$ par $\mathbf{C_{arg}} \subseteq \mathbf{2^{A_{arg}}}$ et $\mathbf{C_{ret}} \subseteq \mathbf{2^{A_{ret}}}$. L'introduction de ces deux domaines nous permet de poser des contraintes sur les arguments et valeurs de retours des services.

Afin de connaître si un service a une instance en cours d'exécution on définit $running : \mathbf{R} \times \mathbf{C_{arg}} \times \mathbf{T} \rightarrow \{\top, \bot\}$. Ce prédicat s'appuie sur la règle suivante :

$$\forall (r, c, t) \in \mathbf{R} \times \mathbf{C_{arg}} \times \mathbf{T} :$$
$$running(r, c)_t \Leftrightarrow \big(\exists (p, x) \in \mathbf{P} \times c : p \in \textit{instancesof}(r, x) \wedge active(p)_t\big) \quad \text{(III.8)}$$

De même on définit aussi le prédicat $last : \mathbf{R} \times \mathbf{C_{arg}} \times \mathbf{C_{ret}} \times \mathbf{T} \rightarrow \{\top, \bot\}$ donnant des contraintes sur le passé d'une requête. En s'appuyant sur la fonction *lastended* définit dans la formule (III.7). On peut le spécifier comme suit :

$$\begin{aligned} &\forall (r, c_a, c_r, t) \in \mathbf{R} \times \mathbf{C_{arg}} \times \mathbf{C_{ret}} \times \mathbf{T} : \\ &\quad last(r, c_a, c_r)_t \Leftrightarrow \big(\exists (x, y) \in c_a \times c_r : lastended(r)_t = (x, y)\big) \end{aligned} \tag{III.9}$$

Ces deux prédicats nous permettent de décrire un état du système en tenant compte de son passé. Ceci nous sert comme base pour la spécification des contraintes liées aux interactions et conflits inter-services dans la couche fonctionnelle. Ça n'est toutefois pas suffisant pour spécifier des contraintes temporelles telles que l'invariance.

Afin de générer notre contrôleur, on souhaite pouvoir spécifier des contraintes qui devront être maintenues en permanence. Le système qu'on surveille a, à chaque instant, plusieurs futurs possibles sans que l'on puisse déterminer à l'avance lequel est le plus probable. En effet l'évolution du système dépend fortement d'évènements dont l'occurrence n'est pas connue a priori par notre contrôleur. En contrepartie, on a une connaissance précise des effets d'un évènement sur l'état du système.

Les logiques modales permettent de spécifier aisément des propriétés à maintenir. Cette extension de la logique propositionnelle introduit les opérateurs de nécessité ($\Box$) et possibilité ($\Diamond$). Il existe des logiques modales temporelles qui posent des hypothèses sur la représentation du temps. Parmi celles-ci, la logique CTL[2][Emerson 90] représente le temps comme une structure arborescente où les sous arbres correspondent aux futurs possibles. Cette logique correspond clairement à nos besoins.

Définition de CTL

La logique CTL s'appuie sur la structure temporelle $M = \{S, R, AP, L\}$ telle que :

S	est l'espace des états,
$R \subseteq S \times S$	est l'ensemble des relations binaires entre états,
AP	est l'ensemble de propositions atomiques
$L = S \rightarrow 2^{AP}$	est une fonction d'étiquetage des états avec les propriétés qui lui sont associées.

[2]**CTL :** Computational Tree Logic

Le graphe sur M correspondant à un espace de représentation CTL est :

1. acyclique - il ne contient pas de cycles orientés.
2. arborescent - tout nœud a, au maximum, un R-parent (ie $\forall(x, y, z) \in S^3$: $(x, z) \in R \wedge (y, z) \in R \Rightarrow x = y$)
3. un arbre - il existe un et un seul nœud tel qu'il n'ait pas de R-parent et que tous les autres nœuds soient atteignables à partir de celui-ci.

Pour raisonner sur cet espace la logique CTL utilise les quantifieurs **A** (always) et **E** (eventually) immédiatement suivis d'un des opérateurs suivants :

$\mathbf{X}p$	indique que la formule p sera vraie au prochain instant
$\mathbf{G}p$	indique que la formule p est vraie tout le temps (invariance)
$\mathbf{F}p$	indique que la formule p sera vraie au moins une fois dans le futur
$p\mathbf{U}q$	indique que p sera vraie jusqu'à ce que q soit vraie en garantissant que q sera nécessairement vraie à un moment donné
$p\mathbf{W}q$	indique que p sera vraie jusqu'à ce que q le soit. La différence avec **U** est qu'il n'est pas nécessaire que q devienne vraie.

Note : Dans les définitions précédentes, p et q sont des formules CTL ou de logique propositionnelle.

Les quantifieurs **A** et **E** indiquent respectivement que la formule suivante doit être vraie pour tous les chemins possibles ou au moins un des chemins.

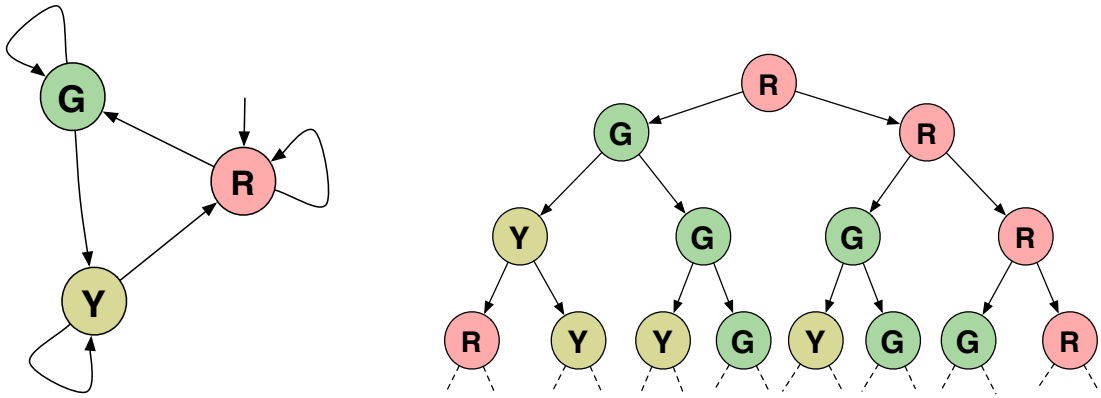
L'ensemble des opérateurs de CTL peuvent être exprimés à partir des opérateurs **EX**, **EG** et **EU**. Les formules d'équivalence sont les suivantes :

$$\begin{aligned}
\mathbf{EF}(p) &\equiv \mathbf{E}[\top \mathbf{U} p] \\
\mathbf{E}[p\mathbf{W}q] &\equiv \mathbf{E}[p\mathbf{U}q] \vee \mathbf{EG}(p) \\
\mathbf{AX}(p) &\equiv \neg\mathbf{EX}(\neg p) \\
\mathbf{AF}(p) &\equiv \neg\mathbf{EG}(\neg p) \\
\mathbf{AG}(p) &\equiv \neg\mathbf{EF}(\neg p) \\
&\equiv \neg\mathbf{E}[\top \mathbf{U} \neg p] \\
\mathbf{A}[p\mathbf{U}q] &\equiv \neg\mathbf{E}[\neg p\mathbf{U}\neg(p \vee q)] \wedge \neg\mathbf{EG}(\neg q) \\
\mathbf{A}[p\mathbf{W}q] &\equiv \neg\mathbf{E}[\neg q\mathbf{U}\neg(p \vee q)]
\end{aligned}$$

Exemple Afin d'illustrer ceci prenons l'exemple simple du feu de croisement dont l'automate de Büchi[3][Thomas 90] est représenté dans la figure III.1(a). La

[3]Un automate de Büchi représente un langage avec objets infinis. Il permet de représenter des systèmes n'ayant pas forcément d'état final et dont les états intermédiaires nous intéressent plus particulièrement.

figure III.1(b) donne une partie de la représentation arborescente de cet automate.



(a) Automate de Büchi (b) Arbre temporel

FIG. III.1 – Un feu de croisement

Si on considère un carrefour avec un feu sur chacune des deux voies, on va chercher à concevoir ce système en respectant les propriétés suivantes :

– $\mathbf{AG}\big(\neg(light_A = GREEN \wedge light_B = GREEN)\big)$ qui exprime qu'il n'existe pas de cas où les deux feux sont verts.

– $\mathbf{AG}\big(light_X = RED \Rightarrow \mathbf{AF}(light_X = GREEN)\big)$ qui exprime qu'un feu au rouge devra passer à un moment donné au vert.

La logique CTL a un pouvoir expressif suffisant pour spécifier des contraintes telles que l'invariance de certaines propriétés (en utilisant l'opérateur **AG**), ou encore l'éventualité de l'occurrence d'une situation (avec l'opérateur **EF**). Par contre, on ne peut pas exprimer un délai maximum ou minimum avant qu'une propriété soit vérifiée. Il existe des extensions – telles que la TCTL [Alur 93] – qui offrent la possibilité d'exprimer les propriétés de façon temporisée (i.e. en introduisant des horloges). Il existe par contre peu d'outils de validation formelle s'appuyant sur cette logique alors qu'au contraire, la logique CTL se prête parfaitement aux techniques de model-checking [Burch 92].

III.2 Génération du contrôleur

Notre contrôleur s'appuie sur le modèle donné dans la section III.1 auquel on ajoute les contraintes du système. Ces contraintes, exprimées en logique CTL, doivent être maintenues durant toute la durée de vie du système afin d'éviter d'atteindre un état indésirable. Le rôle du contrôleur est donc d'empêcher la couche fonctionnelle d'atteindre un état inconsistant tout en respectant au mieux

les demandes issues de la couche décisionnelle. Les contraintes du système étant spécifiques à celui-ci, il faut un moyen de les spécifier en vue de la génération du contrôleur.

Dans cette section nous présentons comment spécifier ces contraintes dans un langage simple basé sur la logique CTL. Nous décrivons ensuite plus précisément les techniques utilisées pour générer le contrôleur correspondant.

III.2.1 L'outil ExoGen

Pour générer automatiquement le contrôleur correspondant aux contraintes du système, nous avons développé l'outil ExoGen[4]. Cet outil prend en entrée les spécifications des modules G°M ainsi que les règles spécifiant les contraintes du système. A partir de ceci il génère les parties spécifiques du R^2C. Les contraintes respectent une syntaxe proche de celle de CTL.

[4] ExoGen : Execution Controler Generator

La spécification des contraintes se fait suivant la syntaxe suivante :

expression	→	`check`{rules}
rules	→	rules rule |
rule	→	ctl_op : formula;
ctl_op	→	`always` | `never`
formula	→	(formula) |
		formula bin_op formula |
		!formula |
		predicat
bin_op	→	`&&` | `||` | `=>`
predicat	→	`running`(run_def) |
		`last`(last_def)
run_def	→	req_name()
		req_name(arg) `with` cstrs
last_def	→	req_name()
		req_name(arg) : ret `with` cstrs
cstrs	→	cstr , cstrs | cstr
cstr	→	var_access domain_def
arg	→	var
ret	→	var
var_access	→	real_var |
		real_var c_access
real_var	→	var |
		`posterval` (var_access)
var	→	`?[a-zA-Z]*`

Dans ce langage les prédicats directement exploitables peuvent exprimer :
– *le passé d'une requête :* On peut exprimer des contraintes sur la dernière instance correctement exécutée de cette requête. Les contraintes porteront sur ses arguments et ses valeurs de retours. Exemple :
`last(setSpeed(?v) with?v>0.1)`
testera si la dernière instance correctement exécutée de `setSpeed` avait bien un argument dont la valeur était plus grande que $0.1m/s$. Ce test s'appuie directement sur le prédicat donné par la formule (III.9) à la page 66.
– *l'existence d'instances en cours d'exécution :* Ce prédicat permet de tester s'il existe une instance d'un service active ou demandée au moment du test. Exemple :
`running(robot_move(?speed) with?speed.linear>1)`
teste s'il existe une instance (ou une demande) du service `robot_move` avec une

vitesse linéaire supérieure à $1m/s$. Ce test s'appuie directement sur la formule (III.8) page 65.

– *Des contraintes sur les valeurs des posters :* Les contraintes sur les arguments – ou valeurs de retour – des services peuvent utiliser la fonction *posterval* pour accéder à la valeur du poster donné en argument. Par exemple le prédicat : `running(track.poster(?p) with posterval(?p).speed in [-1.0 .. 1.0])` surveille, pendant toute l'exécution de `track.poster`, que l'attribut `speed` du poster passé en argument a une valeur comprise entre -1.0 et 1.0.

Pour spécifier un état particulier, on compose ces prédicats avec les opérateurs de logique classique (`!, ||, &&, =>`).

En considérant que, pour un service donné, il ne peut y avoir plus d'une activité démarrée ou finissant à chaque instant, la figure III.2 donne un automate représentant bien l'évolution des prédicats `running` et `last` attachés à un service.

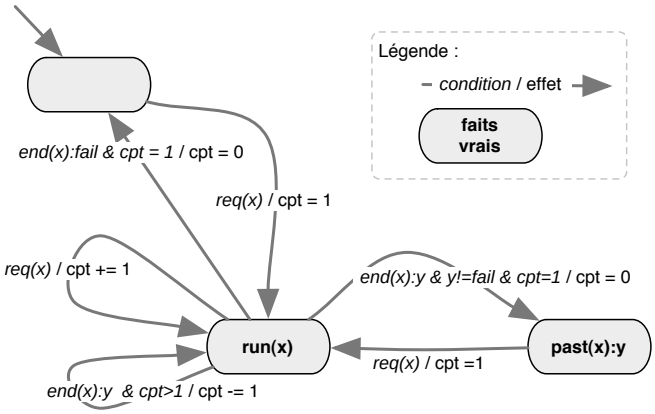


FIG. III.2 – Automate pour un service

On voit sur cet automate que les prédicats `last` et `running` ne peuvent être vrais en même temps. Ceci est dû au fait que l'on considère ici que du moment qu'une instance est en cours d'exécution les effets liés à l'exécution passée d'une instance au même service sont remis en cause et on ne peut donc plus considérer le passé de ce service comme porteur de sens. Il est à noter qu'actuellement dans le langage, les arguments ne différencient pas vraiment le passé des services. Ainsi, quelles que soient les contraintes sur les arguments, s'il existe une instance du service en cours d'exécution alors tout prédicat *last* associé à ce service est fixé à faux. En d'autre terme un service n'a qu'un seul passé et ce indépendamment des valeurs passées en argument. Cette contrainte colle bien aux situations que nous avons rencontrées mais peut montrer ces limites dans le cas où les arguments passés à une demande de service pourraient avoir un sens plus fort.

Un exemple réel est donné par le module `POM` (POsition Manager) développé au LAAS. Ce module fait de la fusion de données afin d'avoir une estimation de la position du robot en fonction de plusieurs producteurs de positions. Un producteur de position exporte sa position via un poster, on attache POM à ce dernier via la requête `POM.addMotionEstimator`, qui prend en argument l'identifiant du poster correspondant. Le lancement de cette requête fait que le module POM devient client d'un nouveau poster. On peut dès lors souhaiter vérifier si on a déjà été connecté à un poster donné et ce, sans avoir à s'occuper si c'est la dernière instance de la requête `POM.addMotionEstimator`. Or, avec le système tel qu'il est présenté actuellement, ce type de test est impossible.

A partir de ces prédicats on ajoute deux opérateurs issus de CTL :

`always(x)` qui correspond à $\mathbf{AG}(x)$. Cet opérateur indiquera les conditions qui doivent être maintenues.

`never(x)` qui correspond à $\neg\mathbf{EF}(x)$. Cet opérateur permet de spécifier les états qu'on souhaite ne jamais atteindre

Exemple

En considérant que nous avons un système avec une caméra montée sur une platine orientable avec les trois services suivants :

`camera_init` Ce service permet d'initialiser la caméra. Il prend en argument une des trois valeurs suivantes `HIGH, MIDDLE, LOW` qui indique la qualité des prochaines prises de vues.

`camera_takeShot` Ce service effectue une prise de vue.

`platine_move` Ce service permet d'orienter la caméra. Il prend en argument une structure avec deux attributs :
- `pan` qui donne la nouvelle position en azimut.
- `tilt` qui donne la nouvelle position en site.

Avec ces trois services on peut spécifier les contraintes suivantes :

```
check {
  never: running(camera.takeshot()) && !last(camera.init(?mode));
  always: last(camera.init(?mode) with ?mode!=LOW) => !(
     running(platine.move(?pos)) && running(camera.takeshot())
  );
}
```

La première contrainte exprime qu'on ne peut effectuer une prise de vue si la dernière initialisation de la caméra ne s'est pas soldée par un succès. La seconde

règle, quant à elle, indique que la platine ne peut bouger lors d'une prise de vue effectuée en un mode différent de `LOW`.

III.2.2 Génération du contrôleur

Maintenant que nous avons un modèle du système ainsi qu'un moyen de spécifier les contraintes nécessaires à son bon fonctionnement, nous pouvons générer le contrôleur correspondant. Nous présentons ici comment nous déterminons les moyens qu'a notre contrôleur pour agir ainsi que la structure de donnée que nous exploitons afin de générer et exploiter notre contrôleur.

En ajoutant la contrôlabilité sur l'automate donné dans la figure III.2, on obtient la figure III.3. ExoGen s'appuie sur cet automate pour générer le contrôleur minimum pour les contraintes données. Un contrôleur minimum empêche le système d'atteindre un état défini comme invalide tout en essayant de perturber le moins possible l'évolution du système. Ceci revient à dire qu'il ne va pas influer sur les évènements qui ne menacent pas la validité des contraintes.

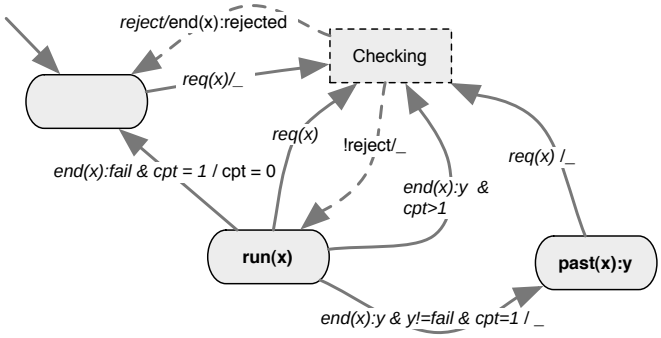


Fig. III.3 – Automate avec contrôlabilité

Dans cette figure on voit que le contrôleur résultant ne pourra que maintenir ou forcer la valeur du prédicat `running` à faux et que bien entendu, il n'a pas de contrôle direct sur la valeur du prédicat `last`. On a donc une contrôlabilité relativement restreinte sur le système.

Afin de générer le contrôleur à partir du modèle de fonctionnement et de l'ensemble de règles/contraintes, nous allons utiliser les techniques issues du "model-checking". Nous utilisons en particulier les OBDDs[5] comme structure de donnée pour représenter les états du système.

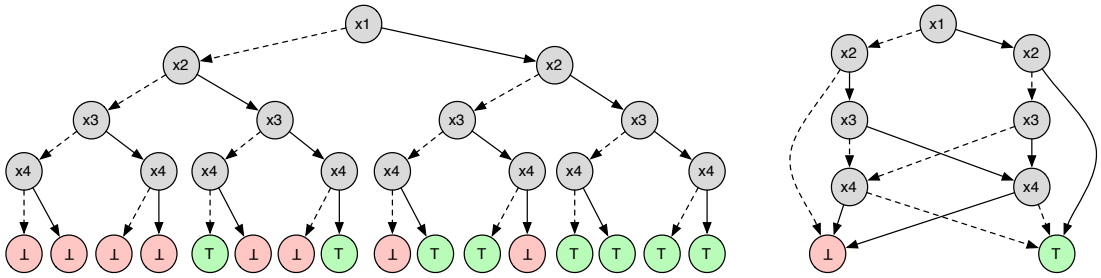
[5]OBDD : Ordered Binary Decision Diagram

Définition des OBDDs

Comme l'illustre la figure III.4, un OBDD [Bryant 86] est une forme compacte de la décomposition de Shannon d'une formule en logique propositionnelle. La décomposition de Shannon d'une formule f sur une de ses variables x_k est le couple de formules $(f_{x_k}, f_{\overline{x_k}})$ telles que :

$$f \Leftrightarrow (x_k \wedge f_{x_k}) \vee (\neg x_k \wedge f_{\overline{x_k}})$$

Ceci revient à, pour la variable x_k, déterminer la fonction f_{x_k}(resp. $f_{\overline{x_k}}$) qui est la réduction de f sachant que x_k est vraie (resp. fausse).



(a) Arbre de Shannon (b) OBDD

FIG. III.4 – Arbre de Shannon et OBDD pour $(x_1 \wedge (x_3 \oplus x_4)) \vee (x_2 \wedge (x_3 \Leftrightarrow x_4))$

Si on applique la décomposition de Shannon récursivement sur les variables x_i de la formule, on obtient une représentation arborescente comme celle donnée dans la figure III.4(a). On peut démontrer aisément que cette décomposition est unique modulo l'ordre des variables [Akers 78] si et seulement si toutes les variables présentes dans la formule sont porteuses d'information (i.e. leur valeur influence la valeur de retour f). La forme de Shannon présente aussi une complexité linéaire sur le nombre des variables pour le calcul de la valeur de la fonction pour une affectation donnée. En effet, une telle opération revient à parcourir un chemin dans l'arbre représentant la formule.

Un OBDD est une représentation compacte de l'arbre de Shannon suivant les règles suivantes :

1. quelque soit le nœud de l'arbre $(x_k, f_{x_k}, f_{\overline{x_k}})$, si $f_{x_k} = f_{\overline{x_k}}$ alors on remplace ce nœud par f_{x_k}. Cette opération de réduction se justifie par le fait que, dans ce cas, la variable x_k n'est pas porteuse d'information et est donc inutile.
2. la deuxième règle consiste à identifier tous les sous arbres isomorphes et de n'en garder qu'un afin que l'arbre devienne un latticiel où il n'existe pas deux sous graphes identiques.

En appliquant ces deux règles on obtient une structure similaire à celle de la figure III.4(b). Cette nouvelle forme apporte les propriétés suivantes :

– Un OBDD est une forme canonique et compacte de la formule pour un ordre de variable donné. Qui plus est les règles de réduction nous assurent que toute variable présente dans un OBDD est porteuse d'information pour la formule que la structure représente.

– L'évaluation de la formule pour une valeur des variables est bornée par le nombre de variables porteuses d'information dans la formule.

– La deuxième règle de réduction des OBDDs assure que le test d'équivalence entre deux OBDDs se fait en un temps constant et ce indépendamment de la taille des deux OBDDs.

Les OBDDs présentent toutefois des défauts dont nous devons tenir compte : même s'ils offrent un encodage compact pour des espaces d'états assez large, cette taille en terme de nœuds, dépend fortement de l'ordre choisi pour les variables. Par exemple, si on reprend l'OBDD de la figure III.4(b) et qu'on inverse x_3 et x_2 dans l'arbre on obtient l'OBDD donné dans la figure III.5. On voit dans cet arbre que le nombre de nœuds dans le graphe est au maximum possible pour un OBDD de 4 variables. Plus généralement cette taille peut passer d'un rapport linéaire à un rapport exponentiel en fonction de l'ordre choisi pour les variables et la recherche de l'ordre optimal est un problème NP-complet. Il existe toutefois des algorithmes qui s'approchent de la taille minimale avec une complexité acceptable lorsque cette opération est effectuée hors-ligne [Rudell 93, Aloul 03].

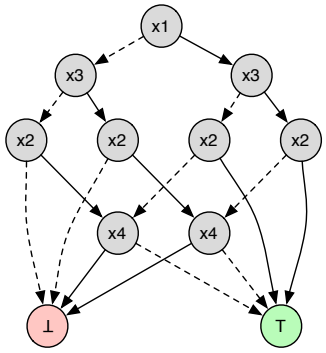


Fig. III.5 – OBDD de la figure III.4 avec un ordre peu efficace

L'algorithme de construction des OBDDs est l'algorithme "if, then, else" – plus communément appelé ITE [Brace 91]. Sachant que la construction d'un OBDD

s'appuie sur les opérateurs de base suivants :

$$\begin{aligned} \top : & \quad \rightarrow OBDD \\ \bot : & \quad \rightarrow OBDD \\ [\_,\_,\_] : & \quad ID \times OBDD \times OBDD \rightarrow OBDD \end{aligned}$$

Le constructeur $[id, tr, fa]$ construit un arbre dont la racine est étiquetée par id et (tr, fa) sont les sous arbres du nœud racine. On garantit que :

$$\forall (id, t) \in ID \times OBDD : [id, t, t] = t$$

L'algorithme ITE reçoit en argument trois OBDDs (a, b, c) afin de trouver l'OBDD correspondant à la formule suivante :

$$(a \wedge b) \vee (\neg a \wedge c)$$

Cette formule peut être formulée de la façon suivante "si a alors b, sinon c" d'où le nom de l'algorithme. La structure sous forme de Shannon se prête bien à cette formulation et permet d'offrir un algorithme simple et rapide donné ci-dessous :

```
ITE(a, b, c : OBDD)
 1  switch
 2     case b = c ∨ a = ⊤ :
 3          return b
 4     case a = ⊥ :
 5          return c
 6     case a = b :
 7          return ITE(a, ⊤, c)
 8     case a = c :
 9          return ITE(a, b, ⊥)
10     case default :
11          n ← plus petite étiquette de a, b et c
12          tr ← ITE(a|_n, b|_n, c|_n)
13          fa ← ITE(a|_n̄, b|_n̄, c|_n̄)
14          return [n, tr, fa]
```

Un des principaux intérêts de cet algorithme est qu'il offre un moyen unique, collant bien à la structure des OBDDs, pour calculer toutes les opérations logiques classiques. En effet, $\neg f$ correspond à $ITE(f, \bot, \top)$ et $f \wedge g$ correspond à $ITE(f, g, \bot)$, on a donc une complétude avec l'opérateur ITE[6].

[6]On rappelle que $(\neg, \wedge)$ est un ensemble d'opérateurs suffisant pour exprimer toute formule logique

Les OCRDs : une extension des OBDDs

Les propriétés des OBDDs sont intéressantes pour pouvoir effectuer des déductions en temps réel. Toutefois, le fait qu'il s'appuie sur la logique propositionnelle est trop limitatif vis-à-vis du pouvoir expressif que l'on souhaite avoir. C'est pour cela que nous avons étendu cette structure avec les OCRDs[7] [Py 02a]. Les OCRDs sont des OBDDs où les variables peuvent avoir des contraintes sur leur domaine associé. Une variable a donc désormais comme étiquetage une expression de la forme $v_i \in d$ où d est un hypercube fixé dans le domaine des valeurs de v_i.

Le principe est ici de générer lors de la construction de l'arbre une partition du domaine de valeur des variables de la formule. On obtient ainsi une structure équivalente à un OBDD où les variables correspondent à un test d'appartenance dans un des éléments de cette partition. Par exemple l'OCRD pour la formule $\big((y \in [0,+\infty[) \Rightarrow (x = \text{OK})\big) \wedge \big((y = 1) \Leftrightarrow (z > 10)\big)$ donnera le résultat représenté dans la figure III.6.

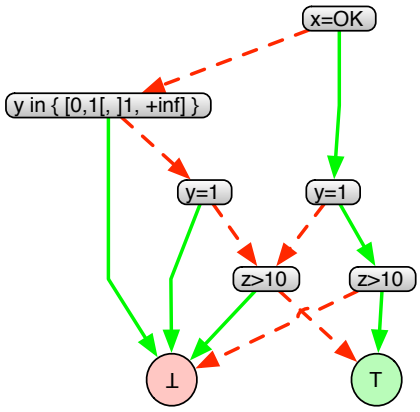


FIG. III.6 – exemple d'OCRD

L'algorithme de construction des OCRDs est assez proche des techniques de construction classique des OBDDs. La différence se trouve dans la création d'une nouvelle contrainte c_i sur une variable v. Pour créer un nouvel OCRD à partir d'une variable $v \in c_i$, on appelle la fonction CREATE-NODE$(v \in c_i, \top, \bot)$ dont l'algorithme est le suivant :

CREATE-NODE$(v \in c : VAR, nthen, nelse : OBDD)$
1 **if** $c = \emptyset$
2 **then return** $nelse$

[7]OCRD : Ordered Constrained Rule Diagram

```
3     else  if ∃ un nœud [v ∈ c, tr, fa]
4             then return [v ∈ c, nthen, nelse]
5             else  if select  c′ tel que (c′ ∩ c ≠ ∅)
                                        ∧(∃ un nœud [v ∈ c′, tr, fa])
6                   then SPLIT-NODES(v ∈ c′, c′ ∩ c)
7                        nelse ← ITE([v ∈ c′ ∩ c, ⊤, ⊥], nthen, nelse)
8                        return CREATE-NODE(v ∈ c/c′, nthen, nelse)
9                   else  ADD-TO-ORDER(v ∈ c)
10                       return [v ∈ c, nthen, nelse]
```

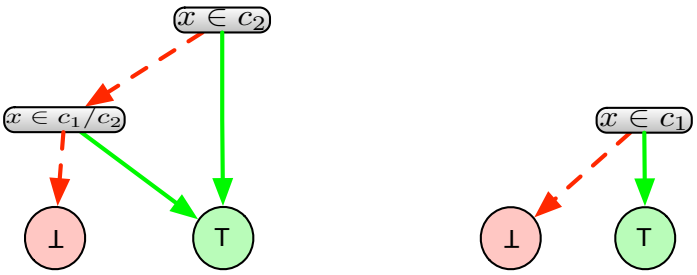
```
SPLIT-NODES(v ∈ c1, c2)
1  if c1 ≠ c2
2    then if c1 ⊈ c2
3           then error "c2  n'est pas inclus dans  c1"
4           else  c3 ← c1/c2
5                 REPLACE-IN-ORDER(c1, {c2, c3})
6                 while ∃ un nœud [v ∈ c1, tr, fa]
7                    do modifier [v ∈ c1, tr, fa] en [v ∈ c2, tr, [v ∈ c3, tr, fa]]
```

La fonction CREATE-NODE construit un OCRD r tel que $r_{v \in c} = tr$ et $r_{\overline{v \in c}} = fa$. On garantit après construction de r que sur l'ensemble des nœuds présents en mémoire :

$$\forall [v_i \in c_1, t_1, f_1], [v_i \in c_2, t_2, f_2] : \big(\exists x; x \in c_1 \wedge x \in c_2\big) \Rightarrow c_1 = c_2$$

Les fonctions REPLACE-IN-ORDER et ADD-TO-ORDER permettent de maintenir l'ordre relatif des étiquetages des nœuds.

L'avantage des OCRDs est que cette structure est équivalente à un OBDD où chaque nœud correspond à un test d'appartenance d'une variable à un sous domaine de son espace de validité. Le fait que ce domaine soit partitionné garantit qu'aucune variable n'est liée à une autre et on se retrouve donc avec un OBDD pouvant être utilisé sur un domaine où l'espace d'états est possiblement infini. On peut par contre critiquer la non remise en cause de cette partition dans la construction.Si on donne en entrée à cet algorithme la formule $x \in c_1 \wedge x \in c_2$ et que $c_1 \subset c_2$, on obtiendra l'arbre donné dans III.7(a) alors que l'arbre III.7(b) aurait été plus compact. On a donc un risque augmenté d'explosion de l'espace d'état dans certains cas. Mais on considérera ici que les formules données en entrée n'auront pas de redondance de ce type.



(a) OCRD résultant (b) OCRD désiré

FIG. III.7 – Problème des OCRDs

Utilisation des OCRDs dans ExoGen

ExoGen utilise les techniques de model-checking décrites dans [Burch 92] appliquées aux OCRDs afin de générer le contrôleur correspondant aux contraintes données. Ce contrôleur s'appuie sur un OCRD final. Cet OCRD représente la formule qui doit être respectée à tout moment afin de ne pas créer d'inconsistance avec les contraintes du système.

le principe de l'algorithme est relativement simple il s'appuie sur la construction classique des OBDDs en ajoutant des fonctions spécifiques pour les opérateurs CTL. Pour ce faire il faut avoir une représentation des transitions d'un état à un autre dans l'automate représentant les transitions entre états. Celle-ci a déjà été donnée dans la figure III.2 (page 71), chaque transition peut être représentée par une formule logique comme nous l'avons effectué dans la section III.1, nous appellerons l'ensemble de ces transitions Tr. A partir de là $\mathbf{EX}(f)$ revient à calculer l'ensemble des états g, qui peuvent mener à f. Si on connaît Tr, cette recherche revient à la formule suivante :

$$Tr(g) = f$$

Cette formule peut être inversée si on prend Tr^{-1}, la fonction inverse de Tr. Ainsi, on peut considérer que :

$$\mathbf{EX}(f) \equiv Tr^{-1}(f)$$

Cette formule nous permet de déterminer l'algorithme pour l'opérateur **EX**. Nous avons déjà indiqué lors de la définition de CTL que tous les opérateurs temporels de cette logique peuvent être exprimés à partir de **EX**, **EG** et **EU**. Nous considérons ici que **EX** est déjà connue. On peut, à partir de là décomposer les deux autres opérateurs avec les formules suivantes :

$$\begin{aligned} \mathbf{EG}(f) &= f \wedge \mathbf{EX}\big(\mathbf{EG}(f)\big) \\ \mathbf{E}[f\mathbf{U}g] &= g \vee \big(f \wedge \mathbf{EX}(\mathbf{E}[f\mathbf{U}g])\big) \end{aligned}$$

Ces deux formules sont exploitées pour effectuer une recherche par point fixe de la formule équivalente à $\mathbf{EG}(f)$ (respectivement $\mathbf{E}[f\mathbf{U}g]$). Les algorithmes qui effectuent cette recherche sont les suivants :

```
EU(a, b : OCRD)
1  res, next : OCRD;
2  res ← b;
3  while ⊤
4     do next ← ITE(b, ⊤, ITE(a, EX(res), ⊥))
5        if res = next
6           then return res
7        res ← next
```

```
EG(f : OCRD)
1  res, next : OCRD;
2  res ← f;
3  while ⊤
4     do next ← ITE(res, EX(f), ⊥)
5        if res = next
6           then return res
7        res ← next
```

ExoGen exploite ces algorithmes pour transformer les formules à valider sous la forme d'OCRDs. De plus il contrôle que la formule finale obtenue ne présente pas de situations en conflit avec les contraintes initiales et sur lesquelles le R^2C n'aura aucun contrôle. Cette analyse revient à vérifier qu'il n'existe pas dans l'OCRD résultat de chemins menant à la feuille `F` et où tout nœud de ce chemin correspond à un prédicat non contrôlable.

La figure III.8 donne l'OCRD obtenu à partir des règles données dans l'exemple de la page 72. On peut voir ici que tout chemin menant à la feuille `F` passe nécessairement par le nœud **running(**`camera_takeshot`**)**. Ce nœud correspond à un prédicat contrôlable que le R^2C peut forcer à faux à tout moment, or on voit que si ce prédicat est faux alors la formule est vraie. On garantit ainsi que toute situation de conflit est contrôlable par le R^2C au moins en agissant sur ce prédicat.

Le rôle d'ExoGen est donc de générer un contrôleur qui manipule cette représentation pour détecter et éviter les états menaçants les contraintes initialement

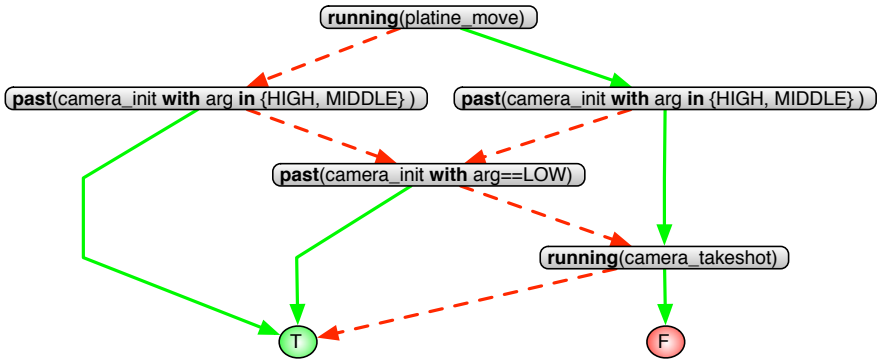


FIG. III.8 – Exemple d'OCRD pour un contrôleur

spécifiées. Pour éviter ces états, il ne peut jouer que sur la valeur des prédicats contrôlables. Il doit aussi respecter le plus possible les demandes issues de la couche décisionnelle.

III.2.3 Propriétés du contrôleur obtenu

Comme nous l'avons présenté en section II.2.2, les deux seuls composants spécifiques du contrôleur par rapport à une plate-forme donnée sont la représentation de l'état du système, qui maintient les connaissances qu'a ce contrôleur sur l'évolution du système, et le contrôleur d'état, qui permet de déduire quelles actions le contrôleur peut faire afin de ne pas violer les contraintes spécifiées. Les OCRDs présentés précédemment offrent les informations nécessaires à la génération de ces deux composants :

– Les étiquettes des nœuds de la structure ainsi que le modèle général donné en figure III.2 permettent d'offrir une traduction des évènements captés par le contrôleur en une représentation symbolique et compacte de l'état du système.

– La structure en tant que telle nous offre un OBDD testant si le système est toujours dans un état valide à partir de la valeur des faits présents dans la base de faits du contrôleur.

Nous présentons ici comment le contrôleur exploite cette structure arborescente pour maintenir le système dans l'espace des états spécifiés comme valides. Cette présentation nous permet de par ailleurs de préciser les propriétés intéressantes de la technique choisie.

Le principe général du contrôleur est relativement simple : celui-ci va chercher à toujours atteindre la feuille terminale maintenant la formule vraie. Pour ce faire le contrôleur construit le prochain état du système en fonction des évènements qu'il vient de capturer et tester si celui-ci est valide. Dans le cas contraire le

contrôleur va regarder quels sont les prédicats mis en cause sur lesquels il a un contrôle suffisant pour modifier l'état futur du système et ainsi rester dans un état cohérent avec les contraintes données. Dans le cadre d'un algorithme de recherche classique, cette recherche a une complexité importante impliquant de possibles retours dans la recherche. Toutefois les OBDDs nous permettent de faire ceci avec une complexité faible et bornée. En effet, on s'appuie ici sur la complexité de la restriction d'un OBDD. Considérons la formule f, représentée par l'OCRD G, la restriction $f|_{\{x_{i_1}=b_1 \ldots xi_k=b_k\}}$ se déduit de G avec une complexité bornée par $O(|G|.log|G|)$, où $|G|$ est le nombre de nœuds de G [Bryant 86]. On exploite cette propriété pour réduire l'OCRD en fixant la valeur des variables incontrôlables. On obtient ainsi une représentation compacte des menaces possibles dans ce contexte ainsi que les moyens de les éviter. On distingue trois types de résultats :

1. la formule est une tautologie : l'état courant ne présente aucune menace et le contrôleur n'a rien à faire.
2. la formule est une contradiction : l'état courant viole les règles données au contrôleur et il n'existe pas de moyen d'en sortir. Il est à noter que l'on peut aisément détecter lors de sa génération si un contrôleur peut atteindre cette situation et donc la corriger lors de la phase de développement.
3. la formule présente tous les prédicats sur lesquels on a un contrôle et le contrôleur n'a plus qu'à sélectionner un chemin dans le graphe correspondant qui mène à la feuille terminale vraie pour rester dans un état valide.

Exemple Prenons l'OCRD donné dans la figure III.8 et considérons que l'état actuel du système est le suivant :

– la dernière fois que le service `camera.init` a été lancé, il l'était avec un argument de valeur `HIGH`.

– une instance de `platine.move` est en cours d'exécution.

– la couche décisionnelle demande le lancement d'une instance de `camera.takeshot`

Avec ces informations le contrôleur peut déduire que le futur état attendu du système peut être représenté par les faits suivants :

$$\begin{array}{lcl} \textbf{running}(\texttt{platine.move}) & = & \top \\ \textbf{last}(\texttt{camera.init with arg} \in \{\texttt{HIGH, MIDDLE}\}) & = & \top \\ \textbf{last}(\texttt{camera.init with arg = LOW}) & = & \bot \\ \textbf{running}(\texttt{camera.takeshot}) & = & \top \end{array}$$

Dans ce cas on a un contrôle possible uniquement sur les prédicats **running**(`platine.move`) – on peut tuer l'instance en cours d'exécution – et **running**(`camera.takeshot`) dont on peut rejeter la demande. Après réduction on se

retrouve avec l'arbre donné en figure III.9. On voit ici qu'il existe deux possibilités pour éviter l'inconsistance :

- forcer **running**(`platine.move`) à faux,
- ou forcer **running**(`camera.takeshot`) à faux.

Pour choisir entre les diverses possibilités, le R^{2}C fixe un poids à chaque nœud. Ces poids peuvent être donnés a priori – on peut par exemple donner un poids plus faible pour les nœuds n'impliquant qu'un rejet de requête – ou dynamiquement – la couche décisionnelle peut donner un poids fort aux services qui lui semblent importants. Le R^{2}C choisit à partir de là le chemin dont la somme des poids est la plus faible. En cas d'égalité il préférera le chemin qui remet le moins en cause l'état du système. Dans notre cas, si les deux prédicats ont le même poids, il choisit de rejeter la demande de `camera.takeshot`.

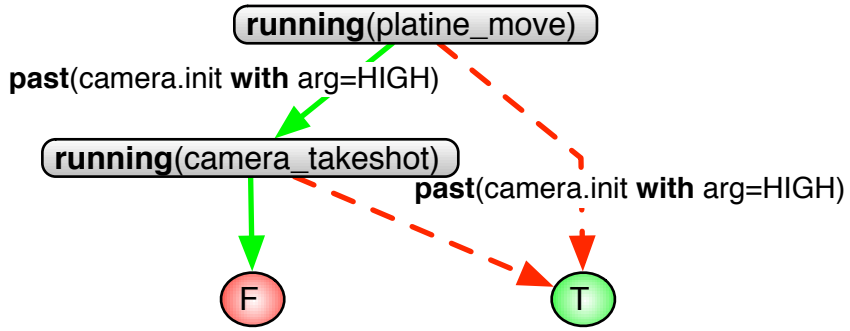


Fig. III.9 – Exemple de déduction du R^{2}C

Par contre, si la dernière instance correctement exécutée de `camera.init` avait eu pour argument `LOW`, alors l'arbre déduit aurait été la feuille `T`. Le contrôleur aurait donc donné l'autorisation de lancer l'instance de `camera.takeshot` sans remettre en cause le reste de l'état du système, cette requête ne menaçant aucunement les contraintes du contrôleur.

On voit donc ici que le contrôleur proposé, en s'appuyant sur les techniques décrites dans cette section, offre des possibilités de contrôle qui semblent suffisantes tout en restant dans des hypothèses synchrones. Ceci peut être garantit du moment que le temps de traitement reste strictement inférieur à la durée minimale séparant l'occurrence de deux évènements non simultanés. Le fait que, grâce aux OBDDs, la complexité de traitement est bornée, nous permet de vérifier et garantir ceci pour une plate-forme et un contrôleur donné. Les expérimentations sur nos robots – décrites dans le chapitre V – ont montré que ceci est le cas pour des systèmes relativement complexes.

Il y a toutefois des limitations sur ce contrôleur. Nous allons maintenant nous pencher sur celles-ci ainsi que des extensions possibles afin d'améliorer les possibilités de contrôle sur des systèmes aussi complexes que ceux qui nous intéressent.

Chapitre IV

Étendre le contrôle

Le contrôleur présenté dans le chapitre précédent est une bonne base pour éviter les états indésirables du système. Il présente toutefois des limitations. Nous présentons dans cette partie les extensions possibles afin d'améliorer ce composant et limiter ainsi les risques de faute du système. Dans un premier temps, nous donnons une approche permettant au R^{2}C de justifier ses interventions de façon précise. Nous enchaînons sur les possibles extensions de notre contrôleur pour lui permettre de repousser les rejets de requêtes afin d'éviter de remettre en cause les décisions de plus haut niveau.

IV.1 Un contrôleur justifiant ses choix

Lorsque le contrôleur force la valeur de certains prédicats, il remet en cause le fonctionnement attendu par la couche décisionnelle. Ceci est toutefois nécessaire, car dans le cas contraire le système aurait été dans un état considéré comme indésirable. Néanmoins, ce bilan non nominal pour la couche décisionnelle oblige celle-ci à remettre en cause ses décisions. Afin que cette dernière puisse reprendre cette contradiction, il faut pouvoir fournir un bilan le plus complet possible.

Le R^2C, tel qu'il a été décrit jusqu'à présent, ne donne pas d'information suffisante aux composants décisionnels pour que ceux-ci puissent faire une reprise d'erreur intégrant les causes de cette dernière. En effet, la seule information

fournie indique le rejet d'une demande de service ou l'interruption d'une activité. Pour une meilleure coopération entre les composants décisionnels et notre contrôleur, il est nécessaire que ce dernier remonte en plus une justification des actions qu'il a effectuées. Cela permettrait d'éviter de retomber dans un cas similaire et, éventuellement, en tenir compte lors des exécutions futures.

IV.1.1 Recherche de la justification

Afin de justifier les décisions de notre contrôleur il faut d'abord pouvoir extraire cette information de la formule qui est maintenue par notre système. La justification d'une décision du contrôleur revient au problème suivant : quelle est la raison pour laquelle il a été nécessaire de changer la valeur d'un prédicat pour maintenir cette formule vraie ?

La difficulté que nous avons ici réside dans plusieurs choix faits pour des raisons de performance. Les OBDDs de par leur structuration sont une entrave à l'extraction de cette information. La factorisation sous forme de Shannon offre une forme compacte de la formule mais la justification est noyée dans un ensemble de données connexes. En effet, si on regarde les nœuds traversés dans un chemin révélant une menace donnée, on a bien sûr les nœuds expliquant celle-ci mais aussi d'autres nœuds qui ne sont pas directement liés à cette conclusion. Suivant la situation, ils expliqueront l'absence ou la présence d'une autre déduction. La décision de rassembler toutes les contraintes en un seul arbre amplifie d'autant plus le problème présenté ci-dessus.

Nous présentons ici les différentes approches que nous avons étudiées afin de pouvoir extraire les causes réelles d'une action de notre contrôleur à partir de l'OCRD sur lequel il se base.

Retarder la décision pour mieux connaître les causes

Une première approche, dans le but de faciliter la recherche de cette justification, est de contraindre l'ordre des prédicats afin que les non contrôlables soient évalués avant les contrôlables dans l'OCRD géré par le R^2C. Ainsi, on évalue d'abord la situation sur laquelle le contrôleur n'a pas de prise pour ensuite obtenir le sous-arbre contrôlable correspondant. Ce dernier donne l'ensemble des actions possibles afin de maintenir le système dans un état consistant. L'idée est ici d'exploiter les informations non contrôlables du parcours dans le graphe pour expliquer la partie – évaluée au plus tard – donnant les actions déduites.

Cette représentation présente l'avantage de s'approcher d'une forme similaire aux règles de production. Toutefois, après analyse, cette forme ne donne pas plus d'information sur les liens causaux entre nœuds du graphe.

Qui plus est, on sait qu'en général l'OBDD de taille minimale a tendance à avoir les variables liées entre elles dans un voisinage proche. Même s'il existe des contre-exemples à cette assertion, elle est assez fréquemment constatée pour être exploitée dans des algorithmes optimisant la taille de ce type de structure [Hu 93]. Nous avons pu constater ceci expérimentalement : les contraintes que nous spécifions lient souvent des variables contrôlables et incontrôlables. Le choix de séparer ces deux types de variables afin que les contrôlables soient évaluées en dernier conduit à une explosion de la taille de notre arbre. Par exemple lors des expérimentations sur le robot Dala – présentées plus en détail au chapitre V – le nombre de nœuds est de 648 quand on contraint l'ordre de la sorte, alors que celle-ci est de 33 avec un ordre optimal. Un tel écart – pour un graphe dont la profondeur est 28 – indique clairement que cette contrainte sur l'ordre des variables dans notre OCRD peut mener à une explosion combinatoire pour des problèmes plus complexes.

Vers une recherche des explications par analyse structurelle

L'analyse structurelle de l'OBDD afin d'extraire les justifications des décisions prises par le R^2C semble donc plus appropriée. Si une telle analyse peut être effectuée, alors on pourra extraire les liens entre prédicat – par exemple constater que le prédicat X n'est évalué que si le prédicat Y l'est aussi – et, par conséquent, obtenir l'information nécessaire pour expliquer les décisions prises par notre contrôleur.

Une première tentative a été effectuée dans ce sens en s'appuyant sur le taux de vérité d'une formule. Ce taux indique, pour une formule donnée, le ratio d'instanciations possibles vérifiant cette formule. Ainsi, pour une tautologie le taux est de 100% et, au contraire le taux est de 0% pour la formule $\bot$. Ce taux est simple à calculer dans un OBDD à sa structuration. En effet le taux de vérité d'une formule peut se calculer lors de la construction en faisant à chaque nœud la moyenne des taux trouvés dans les deux sous arbres. Ainsi, si on calcule ces valeurs pour l'OCRD donné à la figure III.8, on obtient le résultat donné à la figure IV.1.

En exploitant le taux de vérité pour chaque sous graphe de l'OCRD, on peut espérer avoir des informations structurelles sur ce dernier. En effet, si on considère les figures IV.2(a) et IV.2(b), on constate que ce taux décroît quand on va vers la racine pour une conjonction et, à l'inverse, croît pour une disjonction. Or, les variables qui justifient la formule sont liées par une conjonction.

L'idée est ici de repérer dans le graphe les variables réellement liées entre elles grâce à l'indication que donne ce taux. Ainsi, dans la figure IV.1, cette valeur

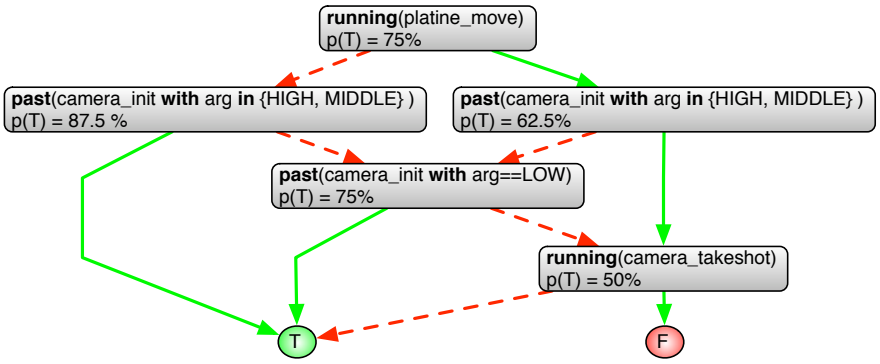
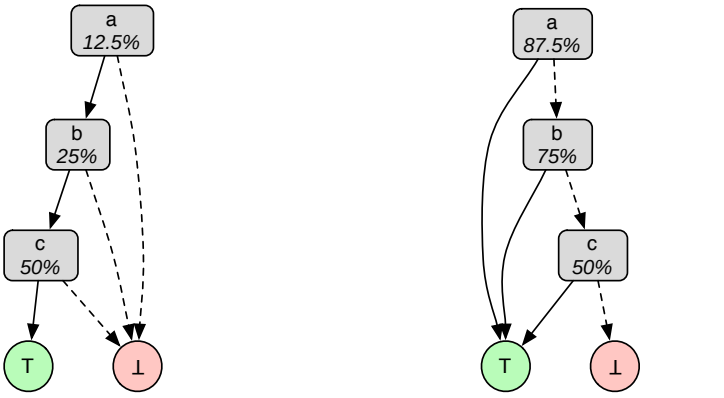


FIG. IV.1 – Un OCRD avec taux de vérité $p(\top)$.



(a) Conjonction $a \wedge b \wedge c$ (b) Disjonction $a \vee b \vee c$

FIG. IV.2 – Cas particuliers pour les taux de vérité.

diminue si **running**(`platine.move`) est vrai (75% vers 62.5%) alors qu'elle augmente dans le cas contraire (75% vers 87.5%). On en déduit que ce test explique une décision si et seulement si il est vrai. Ce lien se retrouve effectivement dans la règle

```
always:
   last(camera.init(?mode) with ?mode!=LOW) =>
       !( running(platine.move(?pos)) && running(camera.takeshot())
```

Même si cette approche semblait prometteuse de prime abord, la difficulté de lui trouver un support formel, ainsi que la mise en évidence de cas semblant indécidables (égalité des probabilités), nous a emmené à écarter cette approche.

Remonter les prédicats non expliqués pour expliquer les choix

Une autre approche pour une analyse structurelle plus poussée a été étudiée en se basant sur l'idée de base suivante. Si on cherche à trouver les liens causaux entre prédicats, pourquoi ne pas commencer par chercher à trouver les prédicats qui ne sont liés à aucun autre. Par exemple, dans la formule $f = a \wedge (b \rightarrow c)$ on voit clairement que pour que f soit maintenue à vraie il est nécessaire que a soit vrai aussi, alors que les valeurs de b et c sont liées. Si on s'appuie sur la structure des OBDDs on peut trouver un algorithme récursif qui va chercher pour chaque sous-graphe ces prédicats non liés et comparer les deux ensembles. Les éléments qu'ils ont en communs ne sont pas liés à la valeur du nœud racine, au contraire ceux qui n'appartiennent qu'à un seul de ces ensembles dépendent de la valeur du prédicat correspondant au nœud courant. Ce traitement est effectué par l'algorithme EXPLAIN décrit ci-dessous.

```
EXPLAIN(t : OBDD)
 1  if t = ⊤ ∨ t = ⊥
 2     then return ∅
 3     else  exp⊤ ← EXPLAIN(t.whentrue)
 4           exp⊥ ← EXPLAIN(t.whenfalse)
 5           selfexplained ← exp⊤ ∩ exp⊥
 6           exp⊤ ← exp⊤/exp⊥
 7           exp⊥ ← exp⊥/exp⊤
 8           if t.whentrue = ⊥
 9              then selfexplained ← selfexplained ∪ {¬t.label}
10              else  if t.whenfalse = ⊥
11                       then selfexplained ← selfexplained ∪ {t.label}
12           return selfexplained
```

Les variables qui nous intéressent plus particulièrement sont $exp_\top$ et $exp_\bot$: ces deux variables donnent les valeurs de prédicats expliquées par le nœud courant en fonction de la valeur du test associé. Cet algorithme, appliqué à l'OCRD III.8, donne le résultat illustré par la figure IV.3.

On voit dans cette figure que se profile un début d'explication de la remise en cause de **running**`(camera.takeshot)`. Une analyse naïve indique que ceci est dû au fait que **last**`(camera.init with arg=LOW)` est faux ou que **last**`(camera.init with arg` $\in$ `{HIGH, MIDDLE})` est vrai. En regardant plus en profondeur, cette explication est tronquée et ne tient pas compte des tests antérieurs.

Afin d'avoir une analyse plus poussée, nous avons ajouté dans l'ensemble des nœuds non expliqués ceux qui correspondent à des tests. Ceci donnait des résultats

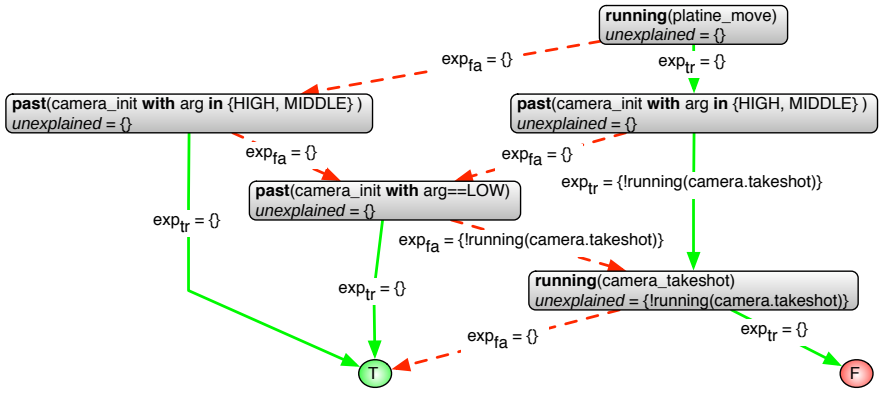


FIG. IV.3 – Résultat de EXPLAIN

proche du résultat attendu. Toutefois, le constat que ce travail correspondait à la recherche d'une forme normale disjonctive[1](DNF) de l'OBDD, nous a poussé à étudier ce qui existait déjà dans la littérature.

IV.1.2 Justification par les impliquants minimaux

Dans les faits, la justification d'un choix du contrôleur est une conjonction qui illustre la sous partie de l'état global du système qui menaçait la formule maintenue par le contrôleur. Les impliquants d'une formule ont une définition assez proche de ceci.

> Soient c et f, deux formules logiques, c étant une conjonction. On dit que c est un impliquant de f si $c \rightarrow f$.

En d'autres termes, si l'impliquant d'une formule est vrai, alors la formule l'est aussi. Ainsi, dans une DNF, toutes les conjonctions qui composent cette représentation sont les impliquants de la formule. Qui plus est, l'ensemble de ces conjonctions est un ensemble d'impliquants couvrant la formule représentée. En effet, un ensemble d'impliquants couvre une formule quand la disjonction de ceux-ci correspond à une DNF de la formule.

> Un impliquant i de f est dit minimal s'il n'existe pas un autre impliquant c de f différent de i tel que $c \rightarrow i$.

Par exemple, pour la formule f telle que :

$$f = x_1 \vee (x_1 \wedge x_2)$$

[1]Pour rappel la forme normale disjonctive d'une formule est la représentation de celle-ci par une disjonction de conjonctions. Par exemple $(a \wedge b \wedge \neg c) \vee (\neg a \wedge c)$.

$x_1 \wedge x_2$ est un impliquant de f, par contre le seul impliquant minimal de f est x_1.

Ainsi, l'ensemble des impliquants minimaux de la formule gérée par notre contrôleur nous donnera une base pour justifier précisément les décisions de celui-ci. Cette justification correspond à l'ensemble des impliquants minimaux qui sont devenus vrais lorsque le R^{2}C a remis en cause la valeur des variables qui menaçaient la consistance du système.

Une technique assez simple pour trouver un ensemble d'impliquants couvrant un OBDD est d'extraire tous les chemins de celui-ci menant à la feuille vraie. Par contre, ces impliquants ne sont pas minimaux. Il faut effectuer un traitement supplémentaire, tel que la méthode de Quine-McCluskey [McCluskey 56], afin d'extraire l'ensemble d'impliquants minimaux à partir de cette DNF.

Dans [Coudert 93], les auteurs présentent une méthode pour extraire les impliquants minimaux d'un OBDD. Cette technique exploite la structure même des OBDDs (décomposition de Shannon) ainsi que la propriété suivante sur les impliquants :

> Considérons une fonction f et la variable x_k. Tout impliquant minimal de $f_{\overline{x_k}}$ (respectivement f_{x_k}) qui est un impliquant de f_{x_k} (resp. $f_{\overline{x_k}}$) est un impliquant minimal de $f_{x_k} \wedge f_{\overline{x_k}}$. Par conséquent il est aussi un impliquant minimal de f.

A partir de ce principe, l'algorithme d'extraction des impliquants minimaux de f se fait via l'appel de la fonction IRRCOVER$(f, f, 0)$:

IRRCOVER$(a, b : OBDD, k : INT)$
1 **if** $a = \bot$
2 **then return** $\emptyset$
3 **if** $b = \top$
4 **then return** $\{\top\}$
5 $g_0, g_1, c_0, c_1, a', b' : OBDD$
6 $P, P_0, P_1 : set$
7 $g_0 \leftarrow a_{x_k} \wedge \neg b_{x_k}$
8 $P_0 \leftarrow$ IRRCOVER$(g_0, b_{\overline{x_k}}, k+1)$
9 $c_0 \leftarrow \bigcup_{p \in P_0} p$
10 $g_1 \leftarrow a_{x_k} \wedge \neg b_{\overline{x_k}}$
11 $P_1 \leftarrow$ IRRCOVER$(g_1, b_{x_k}, k+1)$
12 $c_1 \leftarrow \bigcup_{p \in P_1} p$
13 $a' \leftarrow (a_{\overline{x_k}} \wedge \neg c_0) \vee (a_{x_k} \wedge \neg c_1)$
14 $b' \leftarrow b_{\overline{x_k}} \wedge b_{x_k}$
15 $P \leftarrow$ IRRCOVER$(a', b', k+1)$
16 **return** $P \cup (\{\overline{x_k}\} \times P_0) \cup (\{x_k\} \times P_1)$

Cet algorithme, appliqué à la négation de la formule maintenue par notre contrôleur, nous donne sous une forme minimale les conditions menaçant le système. Ainsi, si on applique ceci sur l'OCRD donné à la figure III.8, on obtient le résultat suivant :

$$\begin{array}{lll} (1) & \text{running(camera.takeshot)} & \wedge \ \text{last(camera.init with arg } \in \{HIGH, MIDDLE\}) \\ & & \wedge \ \text{running(platine.move)} \\ (2) & \text{running(camera.takeshot)} & \wedge \ \neg\text{last(camera.init with arg} \in \{HIGH, MIDDLE\}) \\ & & \wedge \ \neg\text{last(camera.init with arg=LOW)} \end{array}$$

On peut constater que ce résultat est assez proche de la spécification des contraintes donnée à la page 72. Nous pouvons ainsi affirmer que ces deux formules couvrent bien les cas de menaces possibles. Qui plus est, cet algorithme nous garantit de trouver une forme compacte et minimale des contraintes. Ainsi le contrôleur justifiera ces choix avec le minimum d'information nécessaire pour expliquer ce dernier.

IV.1.3 Exploitation des impliquants par le R^2C.

Les impliquants nous donnent directement une information précise et compacte des menaces que peut détecter le contrôleur. Pour trouver la justification de la remise en cause d'un prédicat, il suffit de repérer les impliquants qui sont devenus faux[2] lors du changement de valeur de la variable.

La sélection de ces impliquants peut être effectuée au fur et à mesure de la traversée de notre OCRD en éliminant les impliquants qui ne peuvent être vrais au vu du dernier test effectué. Grâce à cette méthode on se retrouvera à chaque nœud avec une information sur les contextes possibles à cette phase, pour arriver à la fin – avant la remise en cause – avec uniquement les impliquants qui indiquent la raison précise de ces remises en causes. Il suffit donc d'attacher cette explication aux comptes-rendus de choix de notre contrôleur.

Par exemple, si le contrôleur rejette le lancement de `platine.move`, la règle (1) précise que ceci est dû au fait que `camera.takeshot` est en cours d'exécution avec un mode de prise de vue `HIGH` ou `MIDDLE`. Cette règle étant la seule mettant en cause `running(camera.takeshot)`, la justification associée à ce rejet sera directement issue de cet impliquant. Le superviseur a maintenant une information précise sur les causes de l'erreur et peut décider, par exemple, de remettre le lancement `platine.move` à la fin de `camera.takeshot`. Si le rejet avait été remonté

[2]On rappelle que les impliquants que nous exploitons sont ceux de la négation de la formule.

sans justification le même superviseur n'aurait pas pu connaître la cause de ce retour non nominal.

IV.2 Étendre l'horizon pour étendre le contrôle

IV.2.1 Problème lié à la reprise d'erreur

Remonter une justification des décisions prises par le R^2C permet aux composants décisionnels de faire une reprise d'erreur plus fine. Toutefois, on peut s'interroger sur la nécessité de remonter toujours cette reprise vers un plus haut niveau.

Le premier point qu'on doit prendre en compte porte sur les performances des divers composants. Les outils exploités dans la couche décisionnelle explorent un espace de recherche très grand. Il en résulte que la remise en cause de leur décision a un coût algorithmique élevé. Par conséquent, il est souhaitable d'éviter le plus possible de faire une remontée non nominale afin que les performances du système restent acceptables.

Qui plus est, les composants décisionnels ont une faible connaissance sur l'état de la couche fonctionnelle, ainsi que les contraintes nécessaires à son bon fonctionnement. Il en résulte que remonter la reprise d'erreur dans ces derniers ne met pas à l'abri du déclenchement d'un nouveau conflit. Ainsi, dans un système fortement contraint, on se retrouve dans un long échange entre le superviseur et le contrôleur – ce dernier rejetant toutes les alternatives proposées – pour aboutir, dans le pire des cas, à l'abandon pur et simple du but associé à l'action posant problème.

De son côté, le contrôleur a une connaissance précise sur les contraintes de fonctionnement interne de la couche fonctionnelle. De plus, les algorithmes de recherches exploités à ce niveau offrent une grande réactivité. Par contre, son pouvoir décisionnel reste très limité et il n'a pas de réelle connaissance sur les intentions associées à une action. Ces limitations font qu'il y aura toujours des cas où la reprise d'erreur devra nécessairement être remontée vers la couche décisionnelle.

Il existe toutefois des situations de conflits où les connaissances qu'a ce composant sont suffisantes. L'exemple le plus simple est donné par une règle indiquant que deux services ne peuvent être exécutés en parallèle. Dans ce cas le conflit peut être résolu en ordonnant les deux activités. Ce type de résolution de conflits, localisé exclusivement dans le contrôleur, permet de gérer des situations non nominales tout en limitant les remises en causes drastiques des demandes de plus haut niveau.

IV.2.2 Exploiter le modèle pour ordonner les actions

L'approche que nous proposons ici est d'exploiter le modèle géré par le contrôleur – présenté en III.1 – afin que le R^{2}C ait des possibilités d'actions plus étendues que le rejet ou l'interruption d'un service. L'objectif ici est avant tout de permettre au contrôleur de pouvoir gérer des conflits simples ne nécessitant pas de connaissances qu'il n'a pas en sa possession. Ainsi si on prend l'exemple donné à la page 72 et si on s'intéresse plus particulièrement à la règle spécifiant que le service `camera.takeshot` ne peut s'exécuter si `camera.init` n'a pas de passé, notre modèle nous indique que cette règle sera active dans deux situations :

1. `camera.init` n'est pas en cours d'exécution mais n'a jamais été lancée ou la dernière exécution s'est soldée par un échec. Ce qui revient à dire que la caméra n'a pas été initialisée ou pas correctement.
2. `caméra.init` est en cours d'exécution. La caméra est donc en cours d'initialisation.

Le premier cas ne peut clairement pas être géré intégralement par le contrôleur. En effet, celui-ci n'a pas la connaissance suffisante pour fixer l'argument associé à ce service. Une telle connaissance est localisée dans la couche décisionnelle qui est la seule à pouvoir identifier le niveau de qualité que l'image doit avoir pour atteindre le but associé à cette action.

Le second cas par contre est beaucoup plus intéressant à notre niveau. On voit ici, que d'un côté la couche décisionnelle a déjà demandé d'initialiser la caméra et, sans attendre le retour de bilan, demande une prise de vue. Les expérimentations nous ont montré que dans la majorité des cas, ces erreurs sont dues à un problème d'ordre dans la couche décisionnelle. La contrainte de précédence stricte entre les deux activités a été omise lors de la conception. La mise en attente de la demande de `caméra.takeshot` suffit donc à gérer ce conflit.

De plus notre contrôleur a la connaissance suffisante pour pouvoir gérer ce type de conflit. L'automate de la figure III.2 (page 71) indique clairement qu'il existe un état, immédiatement accessible après l'exécution d'un service, où le prédicat `last` de ce dernier est vrai. Cet automate étant exploité par le R^{2}C, une extension l'exploitant plus finement doit nous offrir un mécanisme simple pour gérer ce conflit sans se reposer sur la couche décisionnelle.

IV.2.3 Intégration dans le R^{2}C

Afin de pouvoir implémenter un tel mécanisme dans le R^{2}C on s'appuie sur l'extraction d'impliquants décrite dans la section précédente. Ces impliquants nous permettent de connaître les causes des menaces détectées par le R^{2}C et,

ainsi, de pouvoir exploiter cette information pour détecter s'il est possible ou non de traiter ce problème localement – via, par exemple, la mise en attente de demandes – ou s'il faut remonter un bilan plus classique et se reposer sur la couche décisionnelle.

Pour revenir à notre exemple nous pouvons considérer que l'arbre réduit donné par le contrôleur d'état du R^{2}C correspond à l'arbre donné en figure IV.4. A partir de cette arbre le contrôleur déduit qu'il est nécessaire de fixer

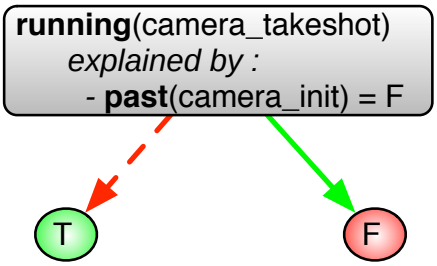


FIG. IV.4 – OCRD réduit pour un camera.takeshot demandé sans init

`running(camera.takeshot)` à faux car, et uniquement car, `camera.init` n'a pas de passé. Une analyse plus en avant de sa base de connaissance lui indique si ce problème peut être géré avec une politique ne remettant pas en cause les demandes initiales. Dans un cas idéal, on a un contrôleur qui effectue une forme d'ordonnancement en fonction des contraintes du système alors que la couche décisionnelle avait fourni un plan avec des parallélismes dangereux pour la fiabilité globale du système. Ainsi la menace détectée par le contrôleur est gérée sans remise en cause des demandes de la couche décisionnelle. On limite ainsi le risque d'invalider le plan de la couche décisionnelle.

Il faut toutefois garder à l'esprit que la mise en attente d'une activité, peut avoir des effets indésirables. Dans le cas où la couche décisionnelle intègre un exécutif temporel – tel que celui présenté dans [Lemai 04] – on peut se retrouver dans la situation suivante :

> L'action mise en attente par le contrôleur peut être considérée comme urgente par les composants décisionnels. Auquel cas, ce délai supplémentaire peut rendre cette action inadaptée au moment où elle sera effectivement lancée.

Il faut garder à l'esprit que, même si elle n'est pas aussi radicale que le rejet, la mise en attente d'une demande reste une remise en cause des décisions de plus haut niveau. Par conséquent, au même titre que pour les autres actions, le R^{2}C doit informer le demandeur que sa demande est reportée. Ainsi, si une telle

décision, prise par le R^2C, va à l'encontre des intentions de la couche décisionnelle, cette dernière en est informée et peut donc intégrer cette modification afin de corriger son plan en connaissance de cause.

Chapitre V

Application à l'architecture LAAS

Nous présentons ici les résultats obtenus lors des expérimentations sur le robot Dala (figure V). Nous décrivons l'ensemble des composants présents sur ce robot et donnons ensuite quelques-unes des règles que nous avons spécifiées dans le contrôleur de ce système. La seconde section donne les résultats et performances des outils développés dans le cadre de cette thèse. Ils mettent en évidence les apports de l'intégration d'un contrôleur sur ce type d'architecture. Ensuite nous mettons en évidence une situation obtenue durant nos expérimentations qui illustre clairement l'influence d'un contrôleur sur les composants décisionnels et nous a mené aux réflexions présentées dans le chapitre IV. Nous donnons finalement d'autres résultats obtenus sur d'autres expérimentations effectuées durant cette thèse.

V.1 Présentation de la plate-forme d'expérimentation

V.1.1 Le robot Dala

Le robot Dala est un robot conçu pour un environnement extérieur. Toutefois, il est aussi équipé de capteurs plus adaptés à un environnement intérieur ce qui lui offre une certaine polyvalence. Cette possibilité de l'exploiter sur terrain accidenté comme en intérieur le rend particulièrement attractif pour nos expérimentations

FIG. V.1 – Le robot Dala

car il intègre un grand nombre de fonctionnalités et modalités d'actions afin de s'adapter au mieux au contexte dans lequel il se trouve. De plus, il a été exploité pour les expérimentations présentées dans [Lemai 04]. Par conséquent, il intègre IxTeT avec son exécutif temporel ce qui lui offre des capacités décisionnelles de haut niveau qui cadrent parfaitement avec notre travail.

Le type de missions qu'est capable de faire ce robot s'inspire de celles des robots d'exploration planétaire Spirit et Opportunity dont la mission sur Mars a commencé en janvier 2004. Dala reçoit des buts sous forme de points où il devra effectuer des photos avec ses caméras. Pour atteindre un point objectif, Dala a deux modalités de navigation :

– la première est adaptée à un environnement non structuré (à l'extérieur). Le robot exploite ici ses caméras stéréos afin de modéliser le relief face à lui et planifie la trajectoire en fonction de la traversabilité du terrain.

– la seconde est plus adaptée à un environnement plan et structuré comme dans un bâtiment. On s'appuie ici sur un capteur laser, situé à l'avant du robot, qui indique les obstacles présents sur le demi-plan face au robot ainsi que leur distance. On planifie ensuite le chemin qui permet de se diriger vers le but tout en évitant les obstacles.

Afin de naviguer correctement, il est nécessaire de pouvoir se localiser. Ainsi Dala est équipé de capteurs sur chacune de ses roues qui permettent de connaître par odométrie le déplacement relatif du robot. Dala ayant un système de déplacement de type "char d'assaut"[1], il a tendance à patiner dans les virages ce qui donne

[1]On entend ici par char d'assaut que les roues de Dala ne sont pas orientables. Ainsi, le seul

une grande imprécision de l'odométrie sur l'orientation du robot. Un gyroscope a été installé pour compenser cette imprécision. De même, en environnement extérieur les données issues des caméras peuvent être traitées afin de détecter le déplacement du robot, ce qui permet de compenser les erreurs de l'odométrie sur un terrain trop meuble.

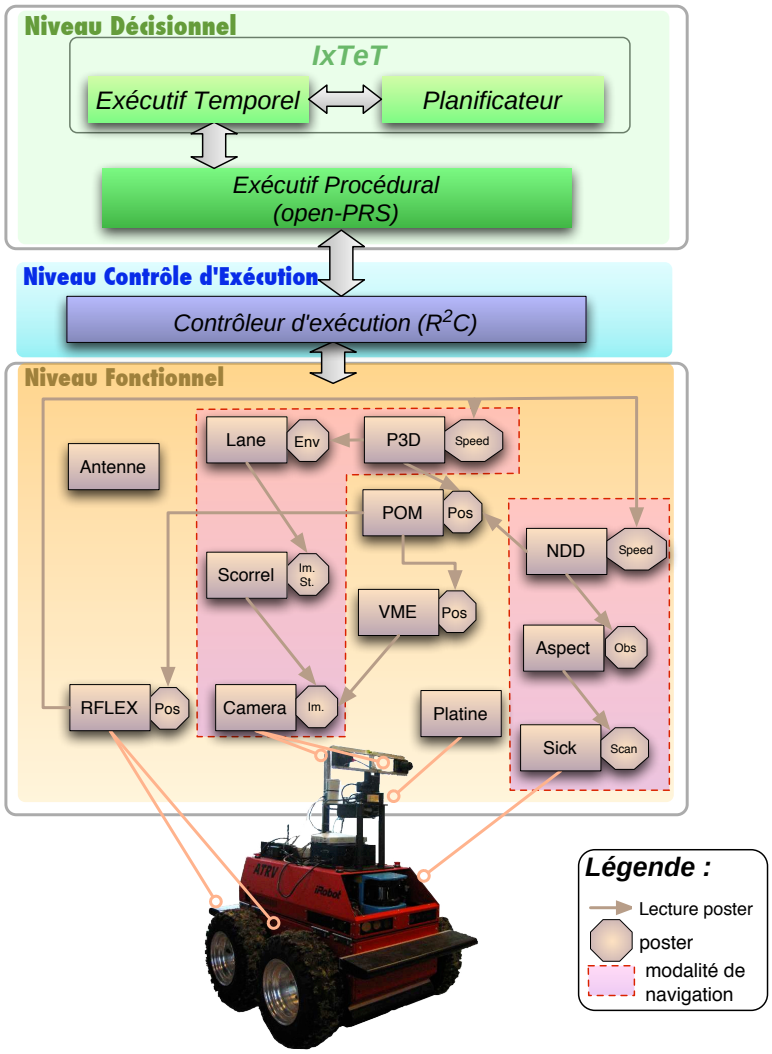


FIG. V.2 – Architecture logicielle de Dala

Pour exploiter ces informations et exécuter les missions de manière autonome

moyen de tourner est de créer un différentiel de vitesse entre les roues de droite et celles de gauche

Dala embarque un ordinateur équipé de 2 Pentium IV à 3 GHz où s'exécute l'ensemble des composants de l'architecture LAAS illustrés dans la figure V.1.1. Les modules présents dans la couche fonctionnelle sont les suivants :

`rflex` est le module qui gère les fonctions d'origine du robot. On y trouve en particulier la commande en vitesse des roues, la production des données odomètriques. Pour des raisons de facilité, ce module gère aussi le gyroscope qui corrige les données sur l'orientation du robot. Ces données sont exportées via le poster `rflex.Pos`.

`camera` permet d'utiliser le banc stéréo du robot. Ce module permet d'effectuer des prises de vues à la demande et éventuellement de sauver celles-ci dans un fichier compressé. La dernière paire d'images produite est disponible dans le poster `camera.Im`.

`platine` gère la platine afin d'orienter le banc stéréo. Celle-ci permet d'orienter le banc suivant les axes de rotation site et azimut.

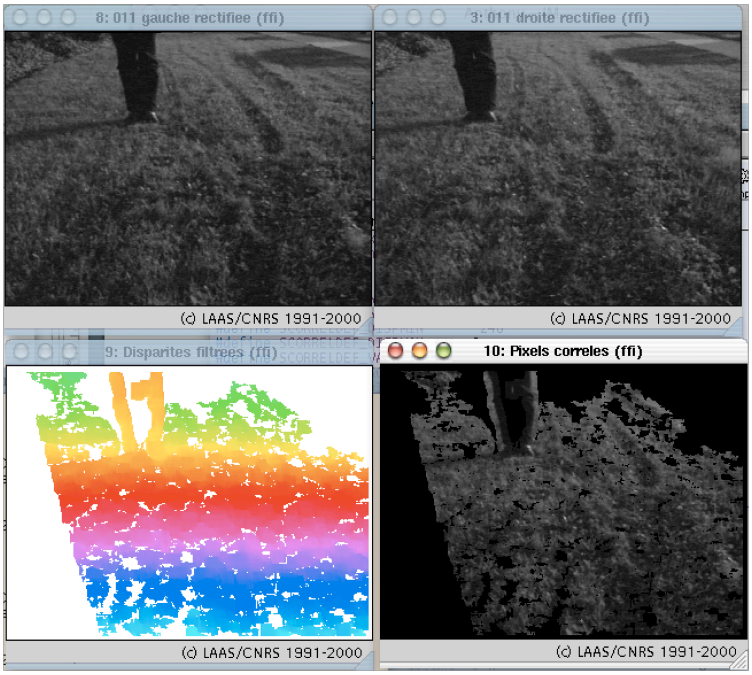


FIG. V.3 – La stéréo corrélation

`scorrel` permet de faire de la stéréo corrélation dense à partir de la paire d'images lue dans `camera.Im`. On obtient ainsi des informations 3D sur l'environnement perçu par les caméras du robot. La figure V.3

illustre les différentes étapes de ce traitement. A partir d'une paire d'images stéréo on calcule la disparité – équivalente à une image de profondeur – pour finalement donner l'image 3D correspondante. Cette image 3D est ensuite exportée via le poster `scorrel.Im.St`.

`lane` intègre les données issues de `scorrel` afin de maintenir un modèle numérique du terrain (MNT) donnant des informations sur son relief comme illustré dans la figure V.4. Cette carte d'élévation est disponible à travers le poster `lane.Env`.

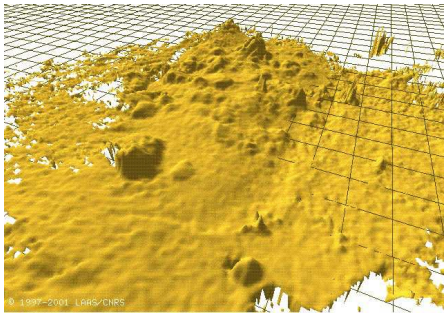


FIG. V.4 – Exemple de MNT produit par `lane`

`vme` calcule le déplacement relatif du robot à partir des images stéréo du robot [Jung 04]. Le principe de l'algorithme est illustré dans la figure V.5. On effectue d'abord de la stéréo corrélation non dense (i.e. uniquement sur certains points de l'image qui sont ici les points de Harris). Ensuite, une corrélation de ces points est effectuée entre les images prises à deux instants successifs. Les points qui ont été corrélés en stéréo et au niveau temporel donnent des informations du déplacement du robot sur les 3 axes en translation (x, y, z) et les 3 axes en rotation (θ, ϕ, ψ) lisibles dans le poster `vme.Pos`.

`p3d` planifie un chemin afin que le robot atteigne une position but donné (x, y) [Mallet 01]. A partir du MNT produit par `lane` et d'un modèle du robot, ce module calcule le chemin le plus stable. Pour ce faire, le système lance des trajectoires possibles sous forme d'arcs de cercles et évalue leurs traversabilité (figure V.6(a)). On itère ensuite suivant une heuristique combinant le risque de rencontrer des obstacles et l'attraction du but pour obtenir le chemin final (figure V.6(b)) qui sera une succession d'arcs élémentaires. Finalement ce module produit les consignes en vitesse correspondantes dans le poster `p3d.Speed` afin qu'elles puissent être lues par `rflex`.

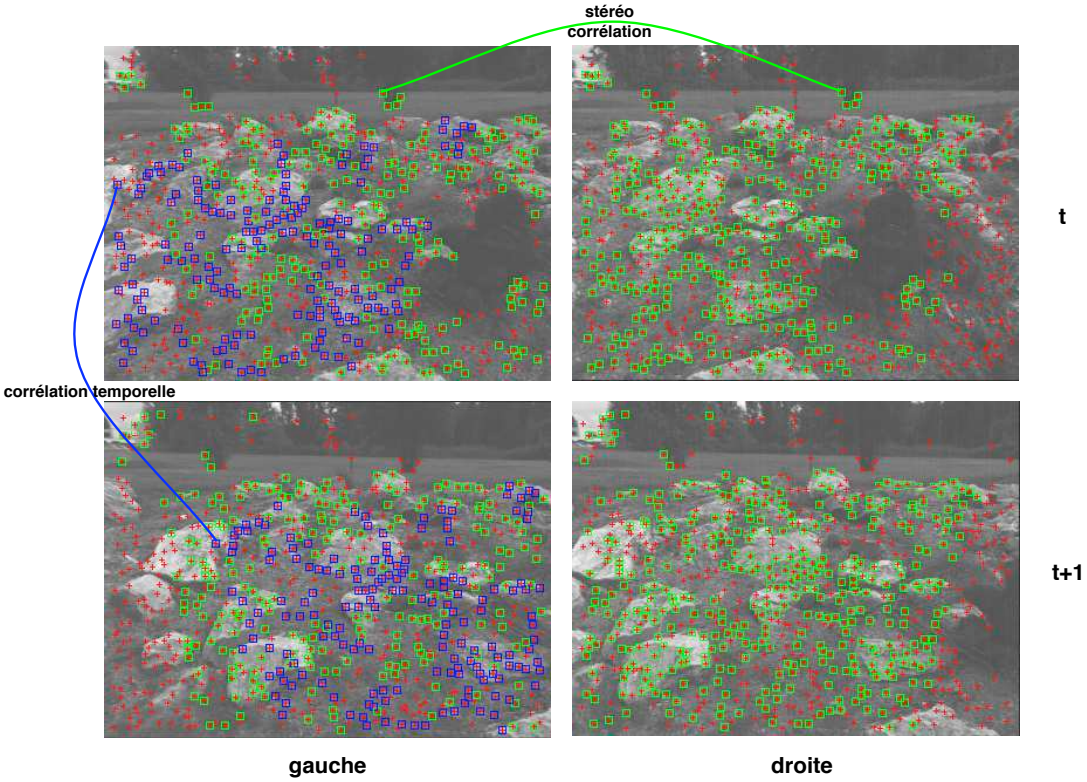
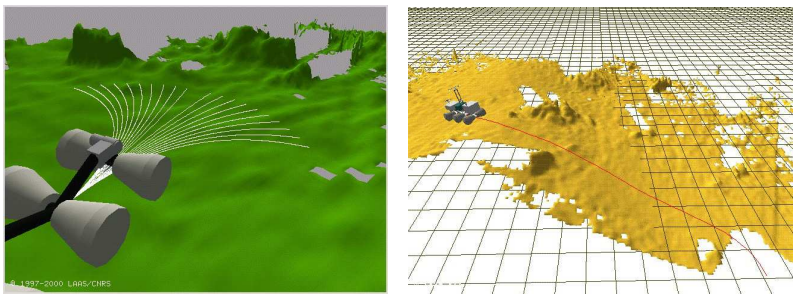


FIG. V.5 – Principe de `vme`



(a) lancer d'arcs (b) chemin final

FIG. V.6 – Recherche de chemin par `p3d`

`sick` offre une interface afin d'exploiter le capteur laser du robot. Ce capteur donne des informations sur la distance et la position relative d'obstacles sur le plan du capteur. Le résultat donne un ensemble de points (obstacles) sur le demi-plan devant le robot comme on peut le voir

à la figure V.7. Ces données sont régulièrement mises à jour dans le poster `sick.Scan`.

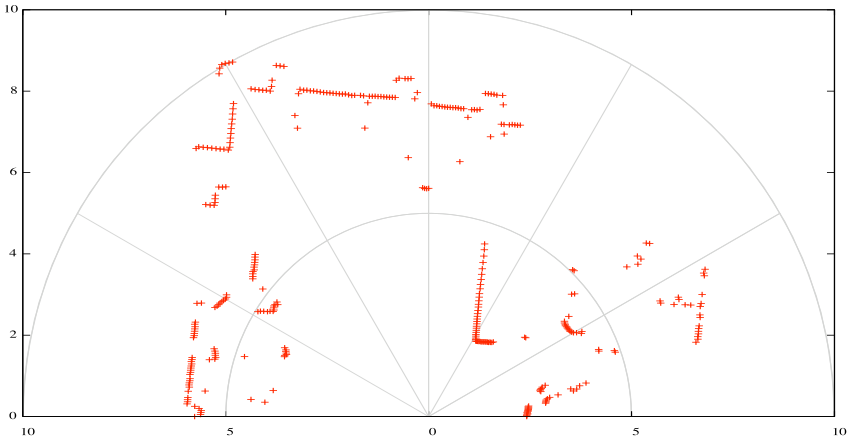


FIG. V.7 – Exemple de données `sick`

`aspect` intègre les données issues du laser afin d'offrir une carte locale de l'environnement dans un voisinage proche du robot. Cette carte locale a une mémoire sur une durée relativement courte (dans nos expérimentations elle est fixée à 10 ms) afin de prendre en compte la dynamique de l'environnement. De plus les nuages de points issus de `sick` sont convertis en segments afin de diminuer le bruit issu des donnés laser et avoir une information plus compacte de l'environnement. Cette carte locale est exportée dans le poster `aspect.Obs` et donne un ensemble de segments qui seront interprétés comme des murs tels qu'on les voit dans la figure V.8

`ndd` propose un algorithme de navigation réactive à partir des données produites par `aspect` [Minguez 04]. Cet algorithme, conduit par le but, détecte les discontinuités entre obstacles et, à partir de celles-ci, découpe son entourage en secteurs appelés vallées. Il sélectionne ensuite la vallée accessible qui le rapproche le plus de son but en minimisant les risques de cul-de-sac. Dans la figure V.8, on distingue clairement 4 vallées délimitées par les obstacles proches du robot. La vallée en vert clair – dans l'axe du robot – correspond au choix fait par `ndd` dans cette situation. Tout comme `p3d`, ce module exporte des consignes en vitesse – destinées à `rflex` – via le poster `ndd.Speed`.

`pom` fait de la fusion de données afin d'estimer la position du robot. En

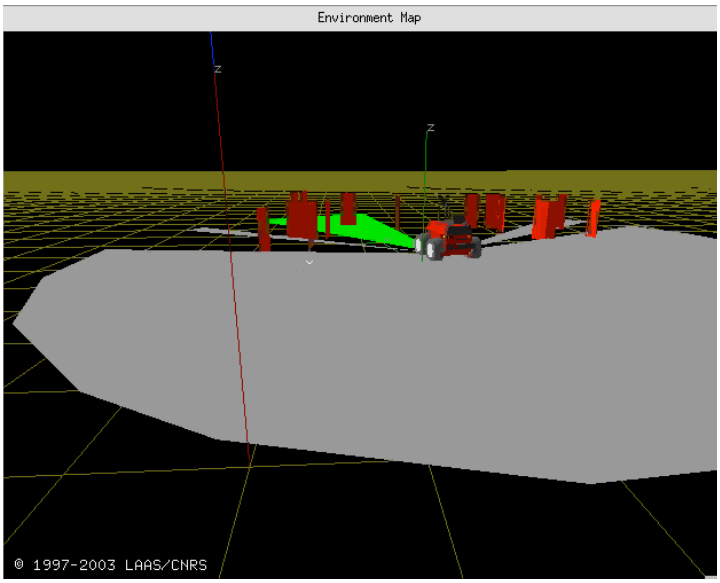


FIG. V.8 – `ndd` en utilisant la carte d'`aspect`

effet, il existe plusieurs moyens d'évaluer la position courante du robot (odométrie de `rflex`, données de `vme`, ...) chacune ayant une précision et une fréquence de production plus ou moins grande. `Pom` a pour rôle d'intégrer toutes ces données afin d'évaluer la position du système avec un taux de confiance associé. De même ce module donne des informations sur la géométrie du système ; par exemple la position de la caméra par rapport au repère du robot en exploitant les données de `platine`. Ainsi dans notre exemple, `Pom` lit les posters `rflex.Pos` et `vme.Pos` en tenant compte de leur fréquence de production. Il peut ensuite produire une estimation de la position réelle du robot qui sera accessible aux autres modules par le poster `pom.Pos`.

`antenna` est un module qui permet de simuler une communication avec un satellite. Ce module n'est pas rattaché à un composant physique et est présent uniquement pour les besoins de la démonstration. Il gère les communications en tenant compte de fenêtres temporelles fixées initialement.

On trouve ici des chaînes de dépendances complexes. Si on revient sur la figure V.1.1, on constate que la majorité des modules accède au moins à un poster d'un autre module pour fournir ses services. Si on garde à l'esprit que la mise à jour de ces posters est associée à une activité du module producteur, on

en déduit que le lancement d'une activité de haut niveau – comme par exemple la planification de chemin de `p3d` – va nécessiter le lancement de beaucoup de services dans des modules tiers (dans notre exemple `camera`, `scorrel`, `lane` juste pour la production du MNT dans lequel `p3d` planifie le chemin).

Du côté des composants décisionnels, on s'appuie sur le système et le modèle présentés dans [Lemai 04]. On y trouve IxTeT comme planificateur avec son extension permettant le contrôle d'exécution temporel. Le domaine utilisé ici se place dans un contexte d'exploration planétaire et prend en compte le temps (durées des tâches, dates d'évènements contingents ...) et certaines ressources (mémoire, disponibilité des caméras ...).

Le type de missions réalisable est illustré par la figure V.9. Le plan est contraint par une durée maximum au bout de laquelle le robot doit être revenu à son point de départ. les buts sont des points d'intérêt où le robot devra effectuer des prises de vues. Il va donc atteindre ces objectifs pour prendre des images qu'il stocke dans une mémoire de faible capacité mais avec un temps d'écriture négligeable. Si cette mémoire est pleine il peut transférer les données vers une mémoire lente mais de très grande capacité. Enfin, le robot doit communiquer avec une sonde durant deux fenêtres temporelles clairement définies où il pourra recevoir de nouveaux buts qui doivent être intégrés dans le plan. Durant cette communication le robot ne peut bouger.

Ce domaine présente plusieurs situations pour lesquelles la réalité lors de l'exécution peut différer du plan initial :

- La durée prévue pour la navigation d'un point à un autre est évaluée a priori par le planificateur sans intégrer les possibles obstacles qui peuvent augmenter ce temps.
- La taille de l'image dépend fortement de la prise de vue (si l'image est complexe son taux de compression sera faible). Par conséquent il est impossible d'anticiper précisément quand cette mémoire sera saturée. Qui plus est, le planificateur ayant une politique optimiste il arrivera forcément un moment où le plan devra être réparé afin d'ajouter une action de transfert des images vers la mémoire lente.
- La possibilité d'ajouter de nouveaux buts en cours de mission remet en cause le plan initial. En effet, il faut intégrer les actions permettant de les satisfaire. Cette remise en cause peut se limiter à une simple réparation du plan – où les nouvelles tâches sont insérées sans modifier l'ordre relatif des tâches du plan initial – ou bien à une replanification complète quand la réparation devient impossible.

L'exécutif temporel aura pour rôle de contrôler ces incohérences afin de réparer

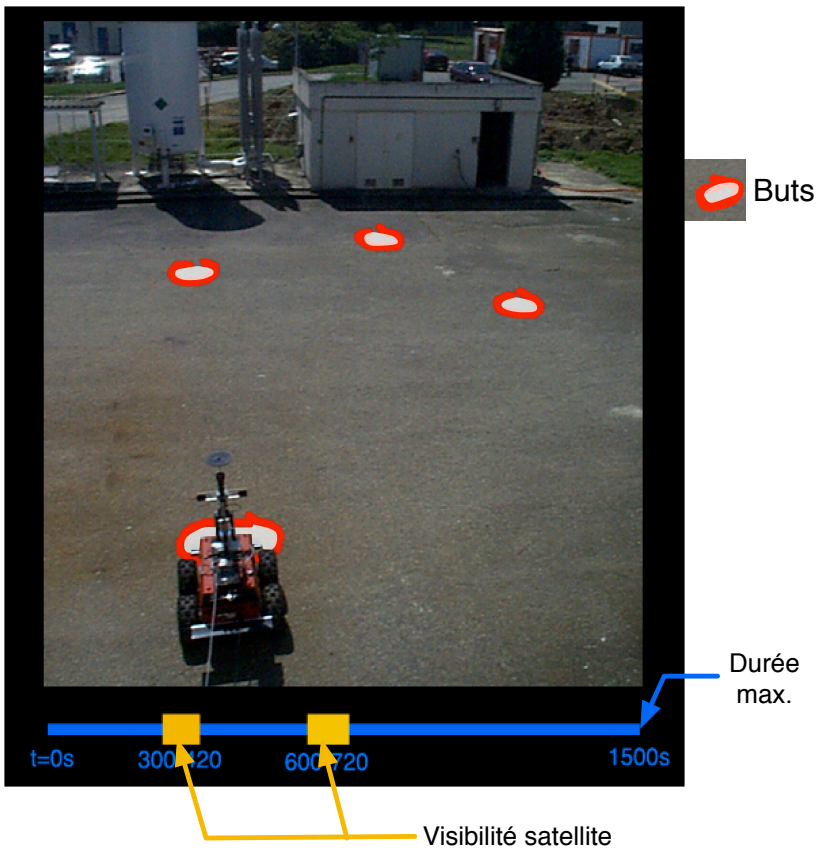


FIG. V.9 – Exemple de mission

le plan. Cette démonstration ayant été conçue afin d'illustrer l'intérêt d'un tel composant, les modèles ont été spécifiés de façon à ce que les situations de conflits se produisent.

Le robot intègre aussi un exécutif procédural basé sur open-PRS. Celui-ci raffine les tâches de l'exécutif temporel en actions de bas niveau. La tâche de navigation est ici un bon exemple pour illustrer la complexité que l'on peut trouver sur le raffinement de tâche. En effet, pour celle-ci on a le choix entre 2 modalités d'actions suivant le type d'environnement dans lequel le robot évolue : dans un environnement fortement structuré où tous les obstacles sont détectables par le capteur laser on se base sur `ndd`, dans un environnement extérieur accidenté on se basera sur `p3d`. Ces deux modalités d'actions ne mettent pas les mêmes modules en œuvre et ont des chaînes de dépendances entre services complexes. On ajoute que, pour les besoins d'IxTeT, on doit pouvoir interrompre la navigation à tout

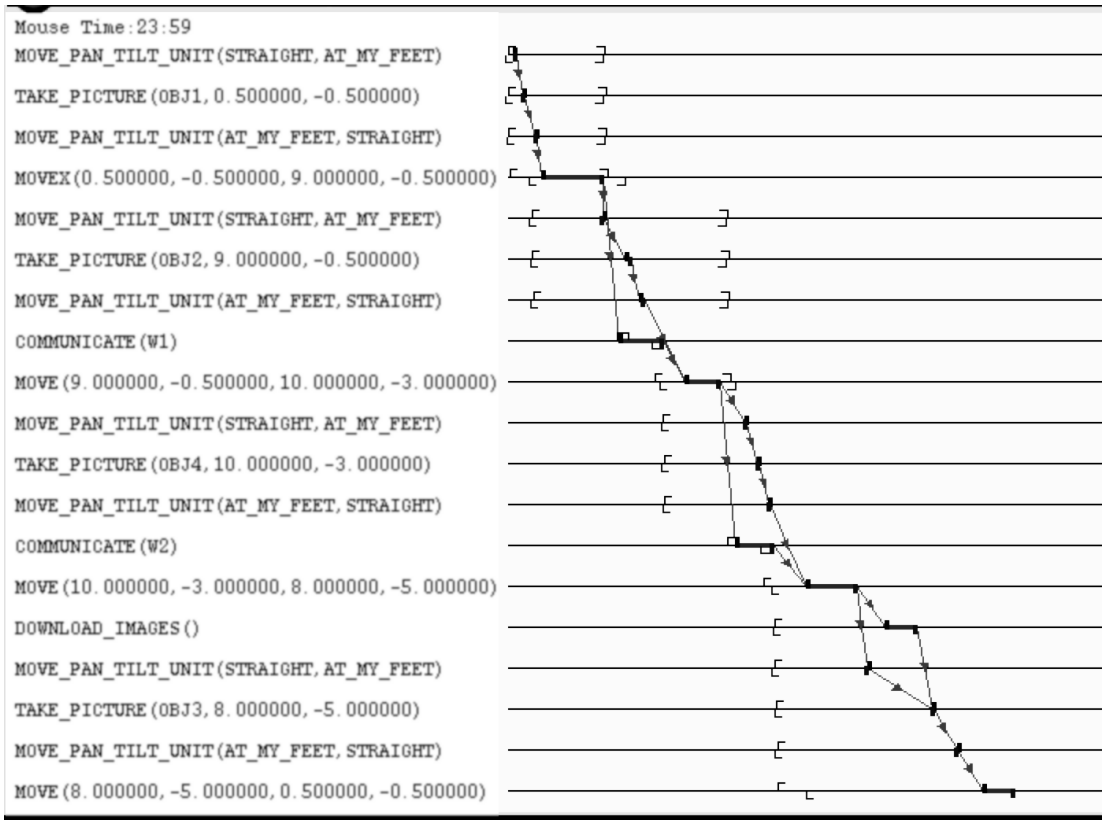


FIG. V.10 – Plan initial pour la mission de la figure V.9

moment indépendamment de la modalité choisie.

Une partie de cette tâche pour le lancement d'un mouvement basé sur `ndd` est illustrée dans la figure V.11. On y voit que juste pour le lancement de ce mouvement, plusieurs activités open-PRS sont lancées en parallèle avec des synchronisations à partir de faits postés par d'autres activités. Un tel mécanisme permet de garder une certaine généricité sur les interfaces de notre exécutif mais augmente la complexité lors de la phase de développement et ce surtout quand on doit appréhender tous les cas de fautes (interruption inopinée d'une activité, problème au lancement, ...). Les autres tâches sont relativement simples (prise de vue à partir des caméras, lancement de la communication, transfert des images, ...).

On peut constater que Dala a un haut niveau d'autonomie. Dans la couche fonctionnelle on trouve un nombre important de modules avec des redondances assurant une certaine robustesse. De plus la couche décisionnelle intègre des composants permettant la planification et l'exécution de plan avec un grand niveau

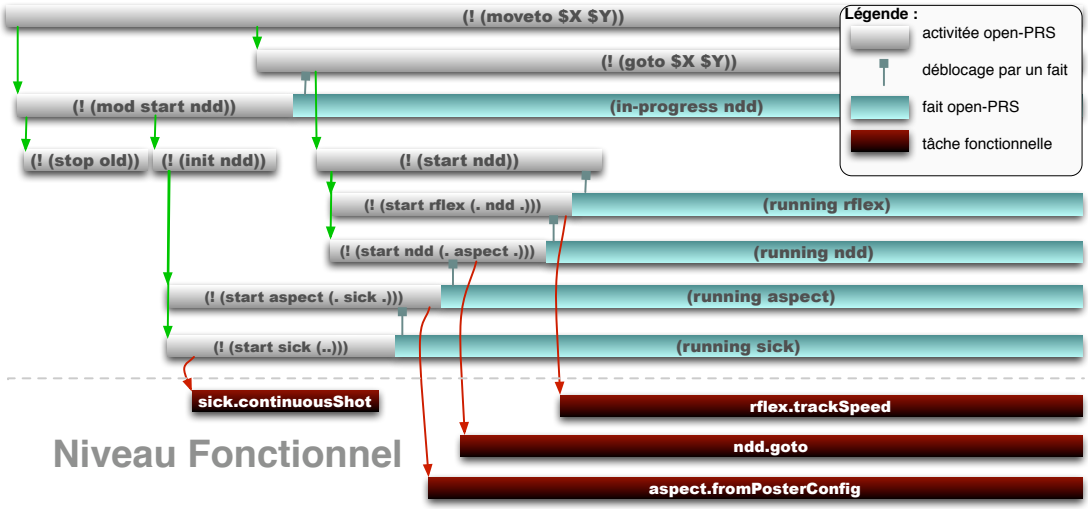


FIG. V.11 – Lancement d'un mouvement commandé par IxTeT dans open-PRS

d'autonomie. La complexité résultante nous offre une plate-forme réaliste pour expérimenter les travaux présentés dans ce manuscrit et en évaluer les résultats.

V.1.2 Conflits présents sur Dala

La première étape à effectuer pour intégrer notre contrôleur est de connaître les situations qui menacent le fonctionnement de notre système au niveau fonctionnel. Pour ce faire nous avons interrogé les développeurs et les utilisateurs des divers modules afin de déterminer, grâce à leur expertise, les situations indésirables sur Dala. Nous présentons ici le résultat de cette étude ainsi que les règles correspondantes exploitées par ExoGen.

Le premier cas qui nous intéresse concerne l'initialisation de `pom`. Elle est relativement complexe et est indispensable pour le bon fonctionnement du robot. En effet ce module est celui qui donne l'évaluation de la position du robot ainsi que les informations sur ces modifications géométriques (position de la platine supportant la caméra, ...). Toutes les informations de ce type sont centralisées dans ce module et vont être exploitées par plusieurs composants (de la couche fonctionnelle et décisionnelle) . Il est donc nécessaire que celui-ci soit correctement initialisé. Pour ce faire, on a défini les deux règles suivantes correspondant à l'initialisation interne du module :

```
#define refME "rflex"
```

```
always: ( running(pom.addME) || running(pom.addSE) )
              => last(pom.SetModel);
never: running(pom.setRefME with arg.name!=refME);
always: running(pom.setRefME)
              => last(pom.addME with arg.name==refME);
```

La première indique qu'on ne peut ajouter d'estimateur de la position (ME : *Motion Estimator*) ou de la géométrie (SE : *Sensor Estimator*) du robot avant d'avoir initialisé le module avec un modèle indiquant les changements de repères nécessaires pour pouvoir intégrer toutes ces données de façon cohérente (`SetModel`).

La seconde et troisièmes correspondent au ME référent utilisé par `pom`. Pour fonctionner correctement ce module à besoin d'un ME qui donne une indication sur la position du robot régulièrement avec une fréquence relativement importante. La précision n'étant pas importante pour ce ME on choisit généralement l'odométrie donnée sur Dala par `rflex`. On a donc une règle indiquant que la requête ne peut être exécutée que si l'argument correspond bien à `rflex`. La seconde règle indique que le ME passé en argument à `pom.setRefME` a dû d'abord être ajouté à la liste des ME gérés par `pom`[2].

`Pom` a un rôle central lors la navigation. En effet, les modules `ndd` et `p3d` ne peuvent contrôler la trajectoire suivie par le robot s'ils n'ont pas une connaissance de sa position vis-à-vis de l'objectif. Pour exprimer ceci nous avons la règle suivante :

```
always: ( running(ndd.GoTo) || running(p3d.Goto) )
          => last(pom.Run);
```

Nous nous intéressons maintenant à la chaîne de dépendances entre services afin de naviguer en utilisant le module `ndd`. Celle-ci met en œuvre plusieurs services des modules `pom`, `sick`, `aspect` et `rflex`. Les règles liées aux dépendances inter modules pour cette modalité sont les suivantes :

```
always: running(ndd.GoTo) => ( last(ndd.SetParams)
                                && last(ndd.SetSpeed with
                                          arg.linear<1.0 ) );
always: running(ndd.GoTo) => running(aspect.AspectFromPosterConfig);
```

[2]Notre compilateur n'ayant à l'heure actuel qu'un support partiel de la logique CTL – il manque en particulier le support de l'opérateur Until – nous avons restreint cette règle au fait que `pom.setRefME` ne pouvait s'exécuter que si la dernière instance correctement exécutée de `pom.addME` avait pour argument le ME référent.

```
always: running(rlex.TrackSpeedStart with arg.name=="nddRef")
          => running(ndd.Goto);
```

La première règle garantit que `ndd` sera correctement initialisé avant d'être utilisé et surtout qu'on a fixé la vitesse maximum produite à $1m.s^{-1}$.

La seconde règle garantit que le module `ndd` ne planifie pas ses trajectoires "à l'aveugle". En effet, pour détecter les obstacles qui entourent le robot, `ndd` s'appuie sur la carte locale fournie par `aspect`. Cette carte n'est mise à jour que si le service `aspect.AspectFromPosterConfig` est en cours d'exécution. Il est donc nécessaire que ce service soit actif quand `ndd.Goto` contrôle la navigation du robot.

La dernière règle garantie que si la commande des roues est attachée au poster de `ndd` alors le service qui met à jour ce poster (`ndd.Goto`) doit être actif.

Toujours autour de la modalité de navigation basée sur `ndd` on a des règles liées au fonctionnement d'`aspect` :

```
#define aspectInput "pomSickFramePos"

always: running(aspect.AspectFromPosterConfig) =>
          last(aspect.SetViewParameters);
always: running(aspect.AspectFromPosterConfig with
                arg.posPosterName==aspectInput)
          => last(sick.SetPomTagging with arg==SICK_TRUE);
always: running(aspect.AspectFromPosterConfig with
                arg.posPosterName==aspectInput)
          => last(sick.ContinuousShot);
```

La première règle contrôle qu'`aspect` est initialisé avant de lancer le service de mise à jour. Les règles suivantes vérifient que `sick` est bien synchronisé avec `pom` et que la boucle locale qui contrôle le laser a bien été lancée (`sick.ContinuousShot`).

Une autre règle présente pour la sécurité limite la vitesse maximale du robot en garantissant que `rflex` ne recevra jamais de consigne en vitesse linéaire supérieure à $1m.s^{-1}$.

```
never: running(rflex.TrackSpeedStart with posterval(arg.name).v>1.0);
```

On voit ici un ensemble de règles illustrant ce qu'on peut définir comme contrôle pour ExoGen. Le reste des règles de Dala est donné en Annexe A. Nous avons spécifié un ensemble de 14 règles ; chacune d'entre elles pouvant contenir de 1 à 5 prédicats.

V.2 Résultats expérimentaux

Les résultats obtenus lors de ces expérimentations illustrent l'intérêt d'intégrer un tel contrôleur dans ce type d'architecture ainsi que la nécessité de prendre en compte les composants décisionnels lors de la conception d'un tel contrôleur.

V.2.1 Performances et utilité du R^2C

La compilation de nos règles pour générer le R^2C et extraire les impliquants de l'OCRD s'est effectuée en $698ms$ sur un mac G4 1GHz. L'OCRD exploité a une profondeur maximum de 28 et est composé de 33 nœuds (figure V.12). On a ainsi une représentation très compacte des 14 règles et ceci nous laisse espérer de pouvoir générer des contrôleurs sur des systèmes beaucoup plus complexes.

Sur Dala, équipé de deux pentium IV à 3GHz, le temps de traitement des évènements[3] par le R^2C est inférieur à $0.4ms$. Ce temps nous permet de garantir nos hypothèses synchrones. En effet, le délai minimum entre deux évènements sur notre système est plutôt de l'ordre de $10ms$ on n'a donc aucune chance qu'un évènement survienne alors qu'on n'a pas fini de traiter les précédents.

Le R^2C a été ajouté à une démonstration déjà existante et son apport fut immédiat. En effet, notre contrôleur a permis de détecter une erreur de codage dans le superviseur lors de la phase d'initialisation des composants fonctionnels. En effet, un problème de synchronisations entre les différents processus parallèles du superviseur faisaient qu'une prise de vue était demandée au module `camera` avant que ce dernier soit totalement initialisé. Le R^2C a immédiatement capturé cette faute et rejeté la requête de prise d'image. De nombreux problèmes dans notre système ont pu être détectés et corrigés de la sorte. Ce fut d'ailleurs la première utilité du R^2C lors du développement de cette démonstration sur Dala.

Cet aspect est d'autant plus intéressant que le superviseur exploité sur ce robot est de par sa conception (plusieurs actions exécutées en parallèle avec synchronisation via des variables partagées) difficile à tester. Le R^2C nous permettait ainsi de pouvoir exécuter le système avec des garanties qu'une erreur dans ce superviseur ne serait pas répercutée au niveau fonctionnel. De plus cette erreur étant capturée on connaissait dès lors la règle violée ainsi que l'état approximatif du superviseur à ce moment là, ce qui nous permettait de localiser le code en cause.

[3]Ce temps est mesuré en temps horloge. En effet, le temps CPU utilisé par le R^2C est inférieur à 1 tic de l'OS et donc impossible à mesurer.

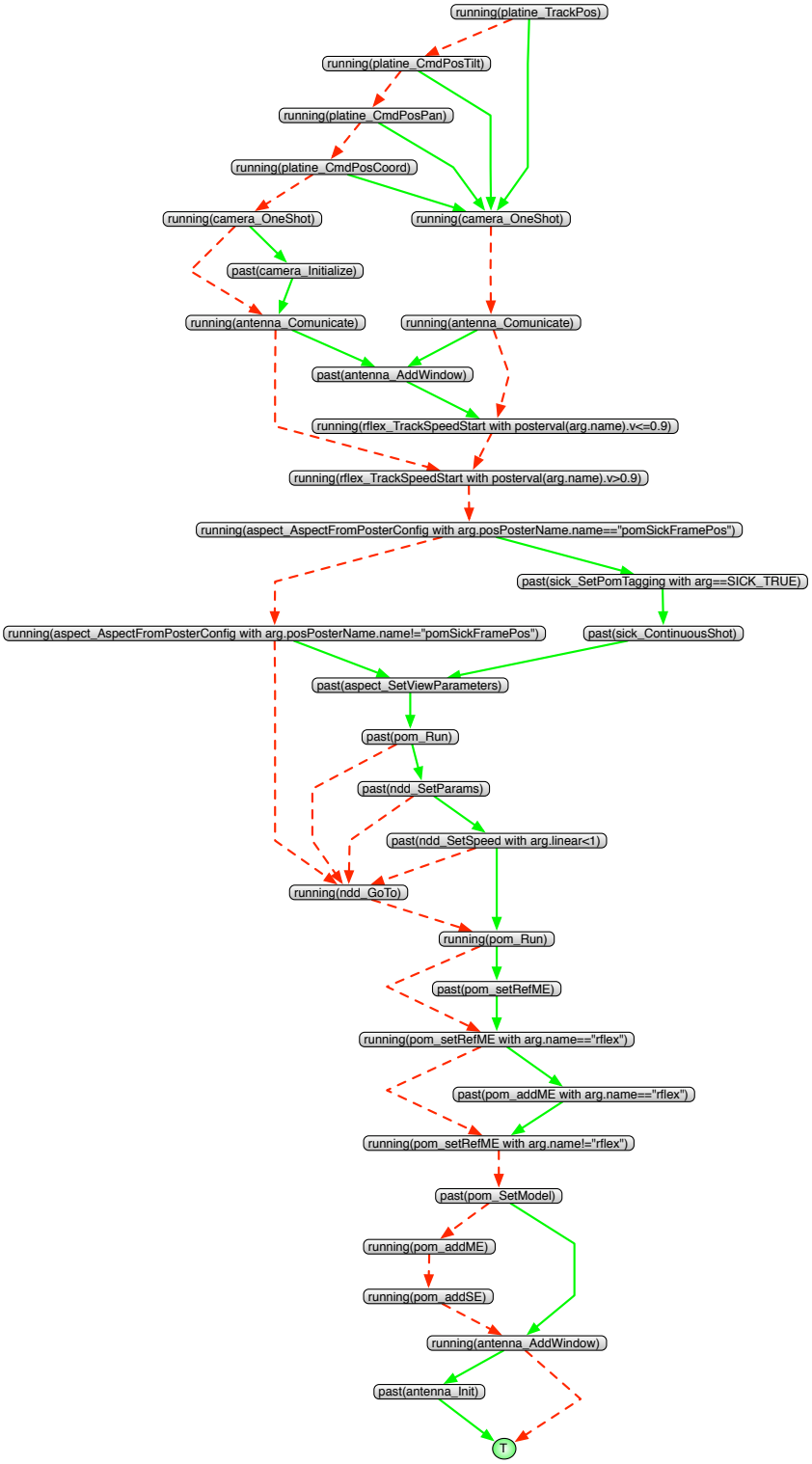


Fig. V.12 – OCRD exploité pour Dala

Toutefois le rôle du R^2C ne se limite pas à celui d'un débogueur. En effet, il doit permettre au système de s'exécuter correctement sans qu'une faute au niveau décisionnel ne se répercute sur le niveau fonctionnel. La figure V.13 donne un exemple d'exécution où le R^2C a détecté et évité trois situations menaçant les propriétés qu'il maintient. La première situation est une faute existante lors de la phase d'initialisation du robot où le superviseur lance une requête afin de déplacer la platine alors que les caméras sont en train d'effectuer une prise de vue. Les deux autres sont des injections de fautes durant l'exécution du plan afin de voir quelles sont les conséquences de la remise en cause de plus haut niveau. La première injection de faute qui consiste à modifier la position de la platine durant le mouvement du robot provoque l'arrêt de la modalité de déplacement, cette action du R^2C est parfaitement reprise par les composants décisionnels. En effet, le planificateur va réparer son plan afin de repositionner les caméras dans le bon axe avant de relancer le déplacement. Dans la seconde situation, où nous avons demandé de lancer une communication alors que le robot est en mouvement, le R^2C a rejeté l'action correspondant à l'injection de faute ce qui n'a pas remis en cause les actions liées au plan et n'a donc pas nécessité de reprise d'erreur au niveau décisionnel.

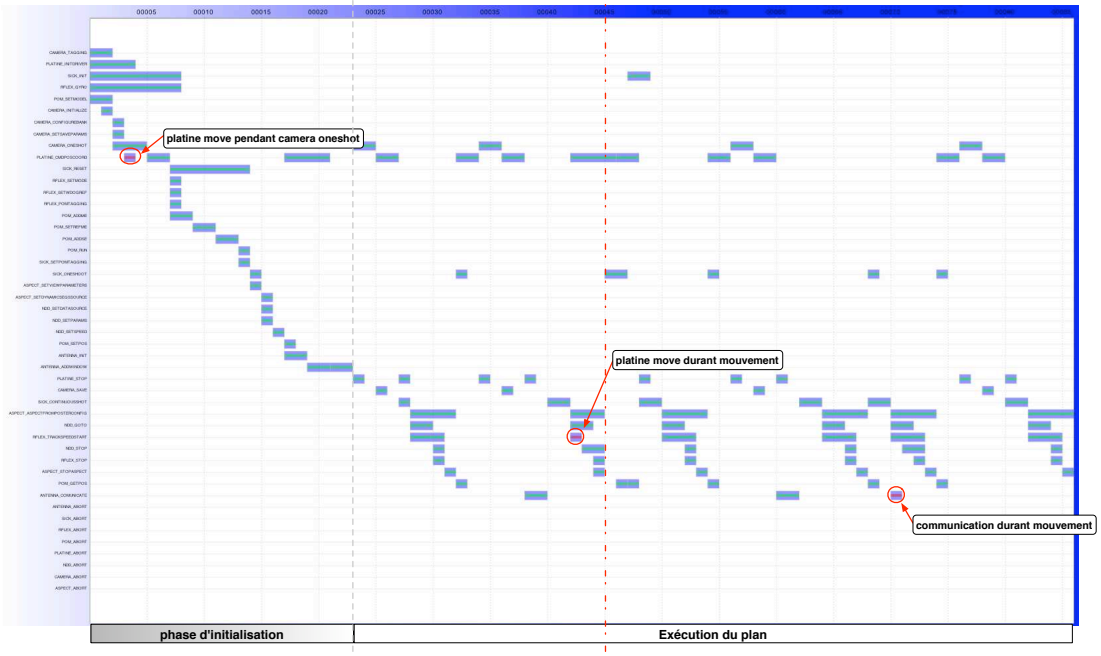


FIG. V.13 – Suivi du R^2C durant le plan de la figure V.10

V.2.2 Prise en compte des composants décisionnels

Les résultats donnés jusqu'à présent illustrent bien le rôle initial du R^2C. En effet, il assure que la couche fonctionnelle respecte les propriétés qu'il maintient en agissant sur les évènements dont il a le contrôle (en rejetant une requête par exemple). On constate aussi que certaines interventions de notre contrôleur peuvent mener à une remise en cause critique des décisions de plus haut niveau (pouvant mener à une replanification complète). Lors des expérimentations nous avons observé des situations menant à un blocage pur et simple du système. Ceci est survenu durant l'exécution d'une navigation avec la modalité basée sur `ndd`. Dans le superviseur l'ordre entre les actions pour lancer un mouvement suivant cette modalité n'était pas suffisamment contraint. Les requêtes `sick.ContinuousShot`, `aspect.AspectFromPosterConfig` et `ndd.Goto` étaient lancées en même temps. Comme on le voit sur la figure V.14, le R^2C détectait immédiatement que ceci était incorrect car `aspect.AspectFromPosterConfig` doit être lancée uniquement après que `sick.ContinuousShot` ait retourné un bilan positif – ce dernier indiquant que le scan laser a bien été lancé. Par conséquent l'ensemble de ces requêtes étaient rejetées. Le problème survenait au niveau du superviseur qui n'ayant que des informations sommaires sur la cause de l'échec de ces actions (le message d'erreur associé était `"R2C rejected"` pour les 3 requêtes) interprétait ceci comme étant un simple échec de `sick` à lancer le scan lié à un problème sur le port série utilisé par ce module. Cette interprétation erronée était remontée à IxTeT qui réparait le plan en conséquence en insérant une action de réinitialisation de `sick` avant de recommencer la tâche de navigation. L'insertion de cette tâche ne corrigeait pas le problème qui se reproduisait ensuite menant à une boucle infinie bloquant le système.

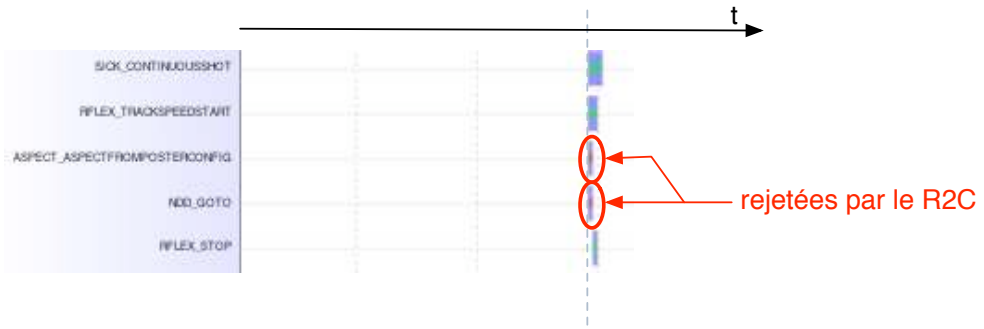


FIG. V.14 – Mauvais lancement de la modalité `ndd`

Ce type de situations nous a confronté au problème de l'influence du R^2C

sur les composants décisionnels et de l'amélioration de l'interaction entre ceux-ci. L'enrichissement des bilans du R^2C avec une explication des décisions prises par ce dernier – comme nous l'avons présenté à la section IV.1 – aurait déjà permis de localiser plus rapidement le problème voire de le corriger.

Toutefois, donner au R^2C la possibilité de retarder les actions menaçant la fiabilité du système (voir IV.2) semble être la solution la plus adaptée à la faute que nous venons de présenter. Dans ce cas, notre contrôleur aurait corrigé lui-même le problème et on aurait ainsi évité une remise en cause du plan. Ceci aurait de plus amélioré grandement la réactivité du système tout en garantissant sa fiabilité.

Nous n'avons malheureusement pas eu le temps de mettre en œuvre ces aspects afin de les tester et les valider. Seule l'extraction des impliquants à partir de l'OCRD et l'injection de ceux-ci dans la structure ont été développées à ce jour. Toutefois nous sommes convaincus qu'une telle approche permettrait un gain appréciable sur le fonctionnement global du système. En effet, notre contrôleur, tel qu'il est implémenté à l'heure actuelle évite qu'une faute issue du niveau décisionnel se propage vers les composants fonctionnels. Toutefois comme nous l'avons illustré ici ceci n'est pas suffisant. Il est nécessaire que notre contrôle ne se limite pas à maintenir la fiabilité du système. Il doit aussi prendre en compte les décisions de plus haut niveau et s'attacher à interférer le moins possible avec ces dernières.

V.3 Premiers résultats sur les impliquants

Nous présentons ici les résultats obtenus sur des expérimentations moins abouties que celles effectuées sur Dala. En particulier le R^2C n'a pas été intégré sur la plate-forme présentée ici. Par contre ces résultats nous permettent de nous pencher plus en avant sur les impliquants présentés – dans le chapitre IV – et de donner une première analyse sur la possibilité d'exploiter cette information en ligne.

V.3.1 Le robot Diligent

Le robot Diligent est un autre robot basé sur l'architecture LAAS. Comme on peut le voir dans la figure V.15, ce robot n'embarque pas de planificateur symbolique. Cette exemple est basé ici sur les travaux présentés dans [Morisset 02]. On y trouve un superviseur basé sur le système Robels qui permet au robot

d'apprendre la meilleure modalité de navigation à exploiter suivant son contexte.

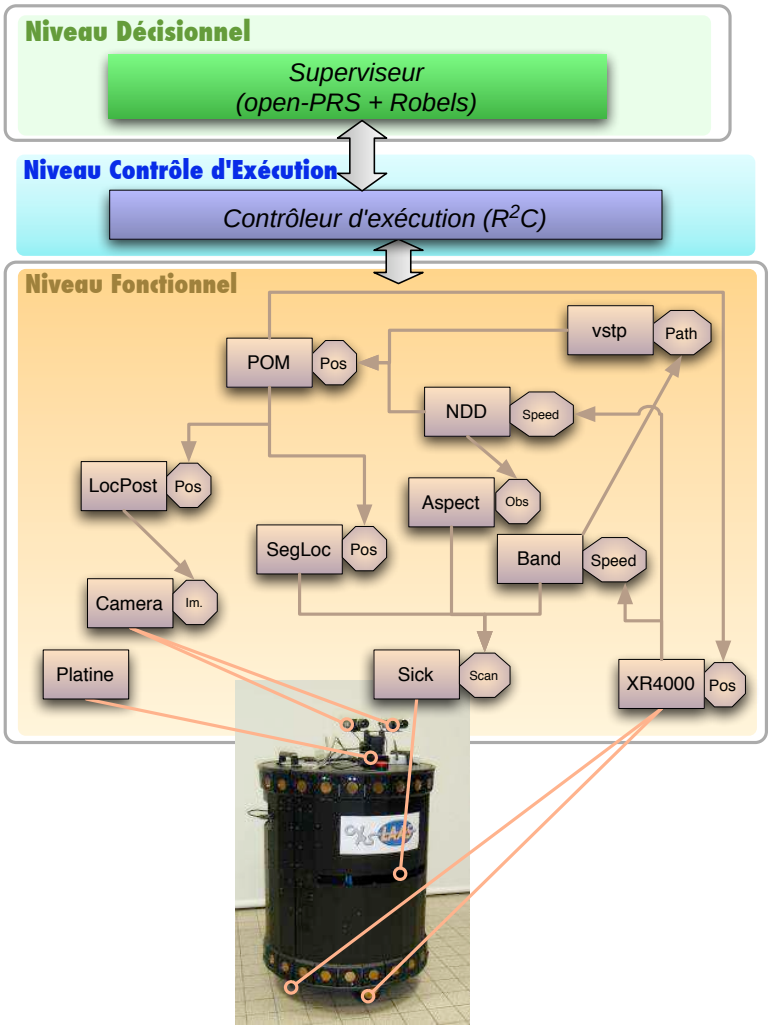


FIG. V.15 – Architecture logiciel du robot Diligent

On peut voir que certains modules présents sur ce robot le sont aussi sur Dala (`pom`, `camera`, `platine`, `ndd`, `aspect` et `sick`). De même le module `xr4000` assure le même rôle que `rflex` sur Dala et ces deux modules partagent la même interface. Ces similitudes nous permettent de réutiliser, sur les deux robots, certaines contraintes sur les interactions entre ces modules. Il faut toutefois faire attention à ce que celles-ci soient bien indépendantes du robot. Par exemple, la contrainte

```
never: running(rflex.TrackSpeedStart with posterval(arg.name).v>1.0);
```

est spécifique à Dala. Diligent est un robot d'intérieur amené à interagir avec des hommes. En conséquence, la vitesse maximum sera plus contrainte que pour Dala. Dans notre cas nous avons fixé cette valeur à 0.8 $m.s^{-1}$. Ce qui nous donne la contrainte suivante :

```
never: running(xr4000.TrackSpeedStart with posterval(arg.name).v>0.8);
```

Les modules nouveaux de Diligent par rapport à Dala sont les suivants[4] :

`vstp` L'acronyme vstp correspond à *Very Simple trajectory planner*. Le rôle de ce module est de planifier dans un environnement structuré et connu une trajectoire pour atteindre un but. A partir d'une carte de l'environnement, ce module planifie une trajectoire qui est une suite de points de passages.Celle-ci est exportée dans le poster `vstp.path`.

`band` Ce module permet d'exécuter une trajectoire avec évitement d'obstacles par déformation. Le principe de déformation est appelé "bande élastique". L'idée est ici de considérer la trajectoire comme un élastique tendu entre le point de départ et le but. Les obstacles exercent une force de répulsion sur cette bande. Ce module est particulièrement adapté aux trajets initialement linéaires avec de possibles obstacles inattendus qui peuvent gêner la progression sans trop la remettre en cause. Ce module est capable d'exécuter ainsi la trajectoire produite par `vstp` en déformant les trajectoires entre chaque points de passages. Les consignes en vitesse sont exportées dans le poster `band.speed`.

`segLoc` Ce module permet de localiser le robot dans un environnement structuré et connu. Pour ce faire, il exploite les données issues de `sick` ainsi que la dernière position connue du robot afin de déterminer la position la plus probable du robot dans la carte. La technique utilisée ici se base sur un filtre de Kalman [Moutarlier 91]. La position évaluée est exportée sur le poster `segLoc.pos`.

`locPost` Ce module permet au robot de se localiser dans son environnement grâce à des amers visuels [Hayet 02]. Les amers sont des rectangles plans du type "poster" (ou affiche). Le module permet de détecter le poster dans une image prise par les caméras et d'identifier si ce dernier correspond à un amer connu auquel est associé une position. La position du robot est ensuite déterminée grâce aux déformations du poster dans l'image perçue.

[4]Dans un but de clarté, nous n'avons pas mis ici l'ensemble des modules présents sur Diligent mais uniquement ceux sur lesquels nous avons spécifié des contraintes.

Nous n'allons pas ici décrire en détail les règles comme nous l'avons fait pour Dala. Toutefois, afin que le lecteur ait une idée des contraintes présentes sur cette plate-forme, le code fourni à ExoGen pour ce robot est donné ci-dessous :

```
check {
 never: running(xr4000.TrackSpeedStart with arg.name.value.v>=0.8);

 always: running(aspect.AspectFromPosterConfig with
                 arg.posPosterName.name.name=="pomSickFramePos")
          => last(sick.ContinuousShot);

 always: running(ndd.GoTo) => ( last(ndd.SetParams)
                               && last(ndd.SetSpeed with
                                         arg.linear<1.0 ) );

 always: running(ndd.GoTo) => running(aspect.AspectFromPosterConfig);




 always: running(aspect.AspectFromPosterConfig) =>
          last(aspect.SetViewParameters);


 always: running(band.MakeBand) => last(band.initBand);
 always: running(band.move) => last(band.MakeBand);


 always: running(aspect.AspectFromPosterConfig with
                 arg.posPosterName.name.name=="pomSickFramePos")
          => last(sick.SetPomTagging with arg==SICK_TRUE);

 always: running(camera.OneShot) => last(camera.Initialize);
 always: running(camera.Acquire) => last(camera.Initialize);
 always: running(locPost.ActivePosterSearch) => running(camera.Acquire);


 always: running(locPost.ActivePosterSearch) => last(locPost.Init);

 always: running(band.MakeBand with arg.name=="vstpTraj")
```

```
    => running(vstp.DynamicTraj);

 always: running(platine.TrackPos with arg.name=="locPostPlatineRef")
   => running(locPost.ActivePosterSearch);

}
```

V.3.2 Résultats et analyse

Pour Diligent l'OCRD résultant – après optimisation de la taille – est de 26 nœuds. La représentation de ce dernier est donné dans la figure V.16. L'extraction des causes possibles d'échec sous forme d'impliquants a été trouvée ici en 2s. Le résultat est le suivant :

1 : ¬running(aspect.AspectFromPosterConfig) ∧ running(ndd.GoTo)
2 : ¬running(vstp.DynamicTraj)
∧running(band.MakeBand with arg="vstpTraj")
3 : ¬last(band.initBand) ∧ running(band.MakeBand)
4 : ¬last(band.MakeBand) ∧ running(band.move)
5 : ¬running(locPost.ActivePosterSearch)
∧running(platine.TrackPos with arg="locPostPlatineRef")
6 : ¬running(camera.Acquire) ∧ running(locPost.ActivePosterSearch)
7 : running(camera.Acquire) ∧ ¬last(camera.Initialize)
8 : running(camera.OneShot) ∧ ¬last(camera.Initialize)
9 : running(locPost.ActivePosterSearch) ∧ ¬last(locPost.Init)
10 : running(aspect.AspectFromPosterConfig)
∧¬last(aspect.SetViewParameters)
11 : running(aspect.AspectFromPosterConfig with
arg.posPosterName.name="pomSickFramePos")
∧¬last(sick.ContinuousShot)
12 : running(aspect.AspectFromPosterConfig with
arg.posPosterName.name="pomSickFramePos")
∧¬last(sick.SetPomTagging with arg=SICK_TRUE)
13 : running(ndd.GoTo) ∧ ¬last(ndd.SetParams)
14 : running(ndd.GoTo) ∧ ¬last(ndd.SetSpeed with arg.linear<1)
15 : running(xr4000.TrackSpeedStart)

Dans la figure V.16, nous avons indiqué sur les arcs correspondants à des choix nécessaires[5] les impliquants qui correspondent à cette situation.

[5]Pour des raisons de lisibilité, nous avons supprimé le nœud terminal ⊥ de ce graphe. Ainsi,

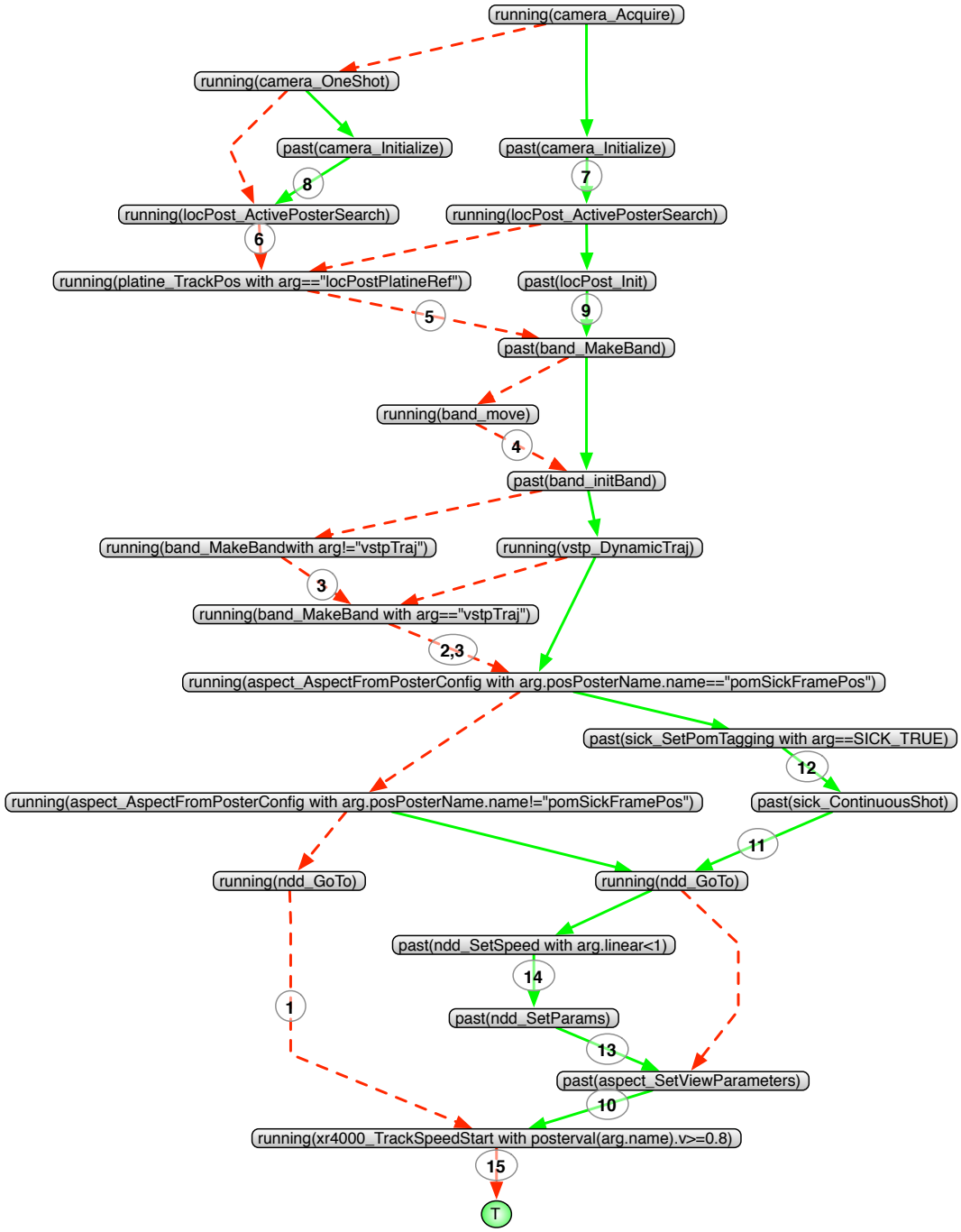


FIG. V.16 – OCRD pour Diligent

les nœuds n'ayant qu'un arc sortant correspondent à des choix nécessaires pour que cette formule soit vraie.

On voit sur cet exemple que, dans la majorité des cas, ce type de nœud est associé à un seul impliquant. Ainsi lors de son exécution, le R^2C pourrait déduire de façon immédiate une des menaces activées. En effet, si lors de l'analyse de l'état on arrive – par exemple – au nœud `[running(band.move)]`, on constate sur ce graphe immédiatement que la seule raison pour laquelle on a aboutit à ce niveau est représentée par l'impliquant 4 :

$$\neg\texttt{last(band.MakeBand)} \wedge \texttt{running(band.move)}$$

Par conséquent si le R^2C devait rejeter où interrompre l'activité `band.move`, L'explication à remonter serait immédiatement donnée par cet impliquant.

Le cas plus délicat est illustré par le nœud `[running(band.MakeBand with arg="vstpTraj")]`. En effet, ce nœud est associé à deux impliquants (2 et 3) et l'impliquant expliquant ce rejet dépend du trajet effectué dans l'arbre. Sur ce graphe on peut voir que la détermination de l'impliquant dépend du nœud traversé juste avant. En effet, si on vient du nœud `[running(band.MakeBand with arg !="vstpTraj")]` alors l'explication est donné par l'impliquant 2 (car on a forcément à ce niveau `¬last(band.initBand)`). Si par contre on vient du nœud `[running(vstp.DynamicTraj)]` alors l'explication est donnée par 3.

On perçoit ici que même si elle n'est pas forcément immédiate, la déduction des explications des décisions du R^2C est réalisable en ligne grâce aux impliquants. En effet, le seul problème que nous avons eu est déterminable grâce à la connaissance du chemin précédemment parcouru dans la structure. Et comme le R^2C effectue ce chemin pour analyser l'état et décider des actions à effectuer, cette information est accessible à moindre coût. L'exemple donné ici est bien sûr relativement simple mais il présente l'idée de base qui apparaît comme étant généralisable aux cas plus complexes.

Chapitre VI

Bilan & perspectives

Le travail présenté dans ce mémoire propose l'intégration d'un contrôleur d'exécution synchrone dans une architecture hiérarchisée. Nous donnons ici un bilan présentant les apports de notre approche mais aussi ses limitations en l'état. Ce résumé nous permet ensuite de présenter les perspectives directement liées à ce travail. Nous concluons enfin sur une discussion plus générale présentant des perspectives sur la mise en place de ce type d'architecture.

VI.1 Bilan général

VI.1.1 Apports du R^2C

Comme nous l'avons souligné au début de ce mémoire de thèse, la fiabilité globale d'une architecture hiérarchisée seule est difficile à garantir. La complexité de ce type de plate-forme inhérente à sa fonction ainsi qu'au milieu dans lequel elle agit rend ce processus lourd et complexe. L'intégration d'un contrôleur d'exécution, à l'interface entre les composants fonctionnels – siège de l'exécution – et les composants décisionnels d'où sont issues les commandes, permet d'offrir des garanties sérieuses sur la fiabilité du système. On contrôle ainsi que les décisions prises par le système ne menacent pas le bon fonctionnement de sa couche fonctionnelle.

Durant la mise en œuvre des outils permettant la spécification du R^2C, nous nous sommes attachés à offrir un modèle et un langage de spécification qui colle au système. Ainsi la partition des domaines sur les données numériques associées aux évènements capturés par le R^2C est gérée par ExoGen. Pour ce faire nous avons spécifié les OCRDs : une structure de donnée qui permet de générer automatiquement cette partition afin d'obtenir un OBDD équivalent. La définition d'une structure gardant les propriétés d'un OBDD nous a permis d'exploiter les techniques de *model-cheking* symbolique et de générer ainsi un contrôleur dont le temps de traitement est borné.

Par contre, nous avons constaté durant nos premières expérimentations que la mise en place d'un contrôleur classique à ce niveau soulevait une nouvelle problématique propre à ce type de système. Dans une architecture hiérarchisée, les actions déduites par le contrôleur vont interférer avec celles initialement demandées par la couche décisionnelle. On se trouve alors avec un conflit entre décisions de haut niveau – dirigées par les objectifs du système – et déductions de notre composant – nécessaires à la fiabilité du système. Afin de limiter ce problème, nous proposons deux solutions complémentaires :

– il est nécessaire que le contrôleur informe la couche décisionnelle des actions qui vont à l'encontre des demandes de cette dernière. Ces informations ne se limitent pas à un simple bilan indiquant les actions effectuées mais doivent aussi indiquer le contexte qui a poussé le contrôleur à agir de la sorte. Ainsi, les composants décisionnels disposent des informations nécessaires pour reprendre la faute correctement.

– le contrôleur doit interférer le moins possible sur les demandes issues du plus haut niveau. En effet, ces demandes sont associées à un objectif connu uniquement par les composants décisionnels. Les actions du contrôleur peuvent aller à l'encontre du plan – dont il n'a pas la connaissance – et donc mener à une remise en cause complète de ce dernier. Pour limiter ce conflit d'intérêt tout en assurant la fiabilité du système, nous proposons de relaxer le modèle de notre contrôleur en lui permettant, quand cela semble "raisonnable", de retarder un évènement qui pose problème. En agissant ainsi, notre contrôleur s'approche d'un séquenceur d'action et offre un contrôle plus souple qui interfère moins sur les décisions de plus haut niveau.

La difficulté pour mettre en place ces solutions vient de la structure même des OBDDs. En effet, pour mettre en place ces nouvelles fonctionnalités il est nécessaire de connaître les liens causaux entre les tests effectués par le R^2C et les choix qu'il effectue. Or, les OBDDs sont des structures de données qui ne mettent pas en évidence cette information. Elles ont même tendance à dissimuler

totalement les liens entre variables. Nous avons donc étudié différentes techniques afin d'extraire cette information. Finalement nous avons exploité un algorithme donné dans [Coudert 93]. Cet algorithme fournit les impliquants minimaux d'une formule qui sont suffisants pour obtenir les justifications d'un choix du R^2C si on connaît le parcours effectué dans l'OCRD qu'il manipule. L'exploitation de cette information n'a pas été encore mise en place dans notre contrôleur mais nous sommes convaincus que sa mise en place ne posera pas de problèmes au niveau des performances et de l'efficacité de notre système.

Notre contribution ne se limite donc pas à la mise en place d'un contrôleur synchrone sur une plate-forme réelle. Nous avons aussi mis en évidence des problèmes liés à l'interaction entre les composants décisionnels. Nous avons proposé des pistes pour la prendre en compte et diminuer les effets néfastes qui en découlent. Grâce à cette extension le R^2C ne se limite pas à un outil mettant en évidence les actions issues du décisionnel qui menacent la fiabilité du système. Il peut rester en place durant la phase d'exploitation. On donne ainsi de meilleures garanties sur la fiabilité du système et ce même sur des situations non anticipées par les développeurs de la plate-forme.

VI.1.2 Limitations de cette approche

Notre approche présente toutefois des limitations qu'il ne faut pas écarter. Certaines sont liées au choix des formalismes exploités pour représenter le système ou exprimer les propriétés de fiabilité, d'autres ne sont pas directement liées au choix de l'approche mais à une nécessité de meilleure intégration de notre contrôleur dans le système.

Concernant les choix de modélisation, nous avons choisi un modèle non temporisé. Ainsi, l'outil ExoGen ne permet pas d'exprimer des propriétés telles que : "La stéréo corrélation ne peut être effectuée qu'avec des images qui datent d'au plus 5 secondes". Le choix de rester à un modèle temporel simple – comme la logique CTL – est principalement dû à une plus grande simplicité et une meilleure maîtrise dans l'état de l'art des outils exploitant ce type de technique. Nous pouvions ainsi focaliser notre travail sur l'intégration et les problèmes résultants. De plus, les propriétés maintenues par notre contrôleur sont relativement simples à traiter. Nous pouvons ainsi offrir un outil agissant en ligne sans que les temps de traitement dégradent fortement les performances du système. Toutefois, l'absence de représentation explicite du temps et de la notion de délais dans notre contrôleur limite son expressivité.

Les possibilités d'actions de notre contrôle restent aussi assez limitées. Nous avons déjà évoqué que le R^2C est incapable de lancer un service qui n'a pas été demandé par les composants décisionnels, il ne peut que bloquer une demande ou interrompre une activité en cour d'exécution. Cette limitation est principalement liée à un choix initial : nous considérons que la prise de décision doit rester localisée dans la couche décisionnelle. Ainsi le rôle du R^2C se limite à contrôler si les demandes du niveau décisionnel ne menacent pas la fiabilité du système et de les rejeter ou interrompre d'autres activités dans le cas contraire. On ajoute que le choix des OCRDs comme base de notre contrôleur joue aussi un rôle dans cette limitation. En effet, les connaissances manipulées par cette structure de donnée sont des domaines. Par conséquent si le R^2C détecte qu'il manque une activité pour que le système reste dans un état viable il va juste donner un domaine des valeurs autorisées pour les arguments. Se pose ensuite le problème de choisir un jeu de valeurs dans ce domaine et notre composant n'a pas été conçu pour pouvoir déterminer ce type d'information.

De par cette limitation le R^2C a un rôle qui se limite au contrôle de la fiabilité du système. De plus, on considère ici que, quand la couche fonctionnelle n'a aucune activité en cours d'exécution, on se trouve dans un état stable. Cette assertion est vraie sur des plates-formes comme le robot sur lequel nous avons expérimenté. Par contre, si on prend un drone, il est nécessaire que les boucles de contrôle qui assurent les commandes de vol soient toujours actives. On parle ici de propriétés de vivacité qui ne sont pas gérées par notre composant. Sur ce type de plate-forme, le développeur qui spécifie les contraintes ExoGen doit bien s'assurer que celles-ci ne menacent pas la vivacité du système.

Au niveau de l'intégration du R^2C dans l'architecture, un problème reste ouvert. Le contrôle lié à l'accès des posters n'est pas satisfaisant. En effet, lorsqu'un poster est modifié par un module le contrôle sur ce poster n'est pas systématiquement effectué. Pire, l'accès à cette valeur n'est pas bloqué et un service client peut lire cette valeur qui n'a pas été validée par le R^2C. Ce problème est principalement lié à un défaut d'intégration de notre contrôleur dans l'architecture LAAS. Il aurait fallu modifier la bibliothèque qui gère les posters afin qu'une modification du poster qui risque de menacer les contraintes du système soit bloquée tant que le R^2C n'en a pas validé la nouvelle valeur.

Nous allons maintenant nous pencher sur les difficultés à spécifier les contraintes du système. Nous avons évoqué dans la section V.1.2 (page 108) que nous avions interrogé les utilisateurs et concepteurs des divers modules afin de rassembler la connaissance nécessaire à la spécification des contraintes du système. Cette tâche est fastidieuse et présente des risques importants liés à une mauvaise compréhension

des informations fournies qui entraînent une spécification de contraintes incohérente du système. Ce problème n'est pas trivial et est d'ailleurs le sujet d'une thèse dans le cadre du projet SAC[1]. Dans ce sujet, l'approche initialement prévue se base sur l'exploitation d'outils classiques d'analyse de la sûreté de fonctionnement (arbres de fautes, AMDEC[2], ...).

VI.2 Discussion et perspectives

Nous avons vu les apports et limitations de l'approche étudiée durant cette thèse. Il est bon de rappeler qu'en dépit de ses faiblesses, le gain apporté – illustré lors des résultats expérimentaux – est indéniable. L'ajout du contrôleur d'exécution sur une plate-forme permet d'offrir des garanties sur le fonctionnement du système tout en y intégrant des composants complexes et difficiles à valider. Nous pouvons ainsi offrir de nouvelles garanties afin de permettre l'exploitation de tels systèmes dans des secteurs critiques.

VI.2.1 Perspectives sur le travail présenté

Les limitations évoquées plus haut permettent de mettre en évidence des perspectives réalisables sur un court terme :

– une première étape serait d'étendre le pouvoir expressif de notre langage afin de permettre la spécification de contraintes plus évoluées. L'extension la plus naturelle serait d'introduire un modèle temporisé en s'appuyant par exemple sur des automates temporisés pour la modélisation de la couche fonctionnelle et la logique TCTL[Alur 93].

– un autre angle d'approche est d'étendre le pouvoir d'action du R^2C afin que ce dernier puisse lancer des services si ceux-ci sont nécessaires au bon fonctionnement du système. Cette extension permettrait à notre contrôleur de jouer un rôle plus important dans l'architecture et de ne plus se limiter au contrôle de la fiabilité du système mais d'étendre ce rôle à d'autres composantes de la sûreté de fonctionnement. Toutefois, une réflexion poussée doit être menée sur cette extension qui donne un rôle qui risque d'interférer avec celui des composants décisionnels.

Il serait aussi intéressant d'approfondir l'interaction entre le R^2C et les composants décisionnels. En effet, cette réflexion ayant débuté vers la fin de la thèse, elle n'a pu être approfondie. Une piste qui semble intéressante serait de permettre aux

[1]SAC : Systèmes Autonomes Critiques

[2]AMDEC : Analyse des modes de défaillances, de leurs effets et de leur criticité

composants décisionnels d'agir sur les choix du R^2C. Il existe déjà la possibilité de fixer des poids à telle ou telle action du R^2C (cf. section III.2.3, page 83) mais cette fonctionnalité n'est pas encore exploitée. On pourrait de toute façon aller plus loin en explicitant ce type d'information. Par exemple une variable testée par le R^2C et mise à jour dans la couche décisionnelle, permettrait aux composants décisionnels de modifier le comportement du R^2C sans que cette influence se fasse aux dépens de la fiabilité du système. Mieux, une telle modification permettrait une meilleure interaction entre ces deux éléments de l'architecture.

VI.2.2 Exploitation des OCRDs pour exprimer les horloges

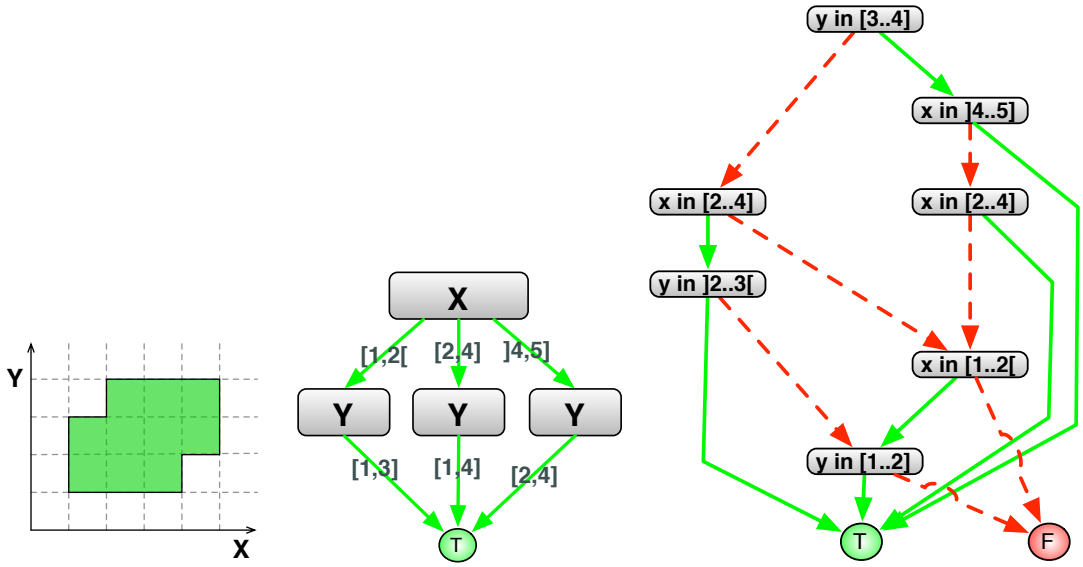
Les OCRDs présentent aussi une exploitation intéressante qui n'a pas été étudiée durant cette thèse. Ils pourraient être exploités afin de représenter les contraintes entre horloges dans un automate temporisé. Cette représentation est une composante centrale des outils de model-checking temporisé.

Si on regarde, par exemple, l'outil UPPAAL il exploite une structure appelée CDD[3] [Larsen 99]. Cette structure représente les contraintes entre les horloges sous une forme inspirée des OBDDs. Comme l'illustre la figure VI.1, cette figure peut être représentée aussi sous forme d'un OCRD. De même pour des contraintes plus complexes de la forme $x \leq y$, il est possible de représenter ceci sous forme d'un OCRD en utilisant le même artifice que pour les CDD qui consiste à représenter la pseudo-variable $x - y$. On obtient ainsi les résultats sonnés dans la figure VI.2. On peut ainsi constater que pour tout CDD, il existe un OCRD équivalent.

Cette première analyse est encourageante. Il reste toutefois à analyser ce point plus en détail afin de vérifier que les algorithmes exploités dans les CDDs sont transposables sur notre structure. De même le gain apporté par l'exploitation des OCRDs pour représenter les contraintes des horloges reste à étudier. Une piste dans ce sens serait sur la canonicité de la représentation. En effet, il semblerait que les CDDs ne soient pas une forme canonique des contraintes. Les OCRDs, par contre, son canonique pour une partition de l'espace donné. Il reste donc à vérifier que cette propriété est bien respectée dans ce contexte et le gain que cela pourrait apporter.

Un autre point connexe à cette étude est lié au constat que la structure même des CDDs correspond à celle que nous exploitons pour représenter les contraintes d'une variable de dimension quelconque. Par contre les CDDs permettent de définir des contraintes plus fines – comme nous l'avons vu dans la figure VI.2.

[3]CDD : Clock Difference Diagram



(a) Domaine de la contrainte (b) CDD (c) OCRD

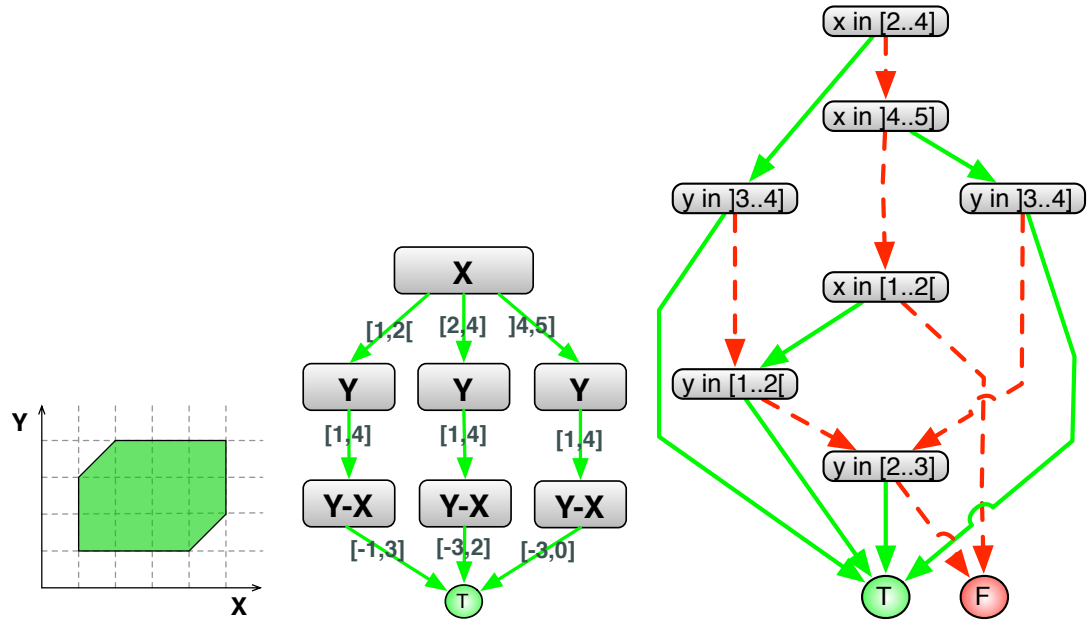
FIG. VI.1 – Contrainte simple entre 2 horloges

L'exploitation des CDDs pour exprimer les partitions des variables associés aux OCRDs augmenterait sûrement le pouvoir expressif d'ExoGen.

VI.2.3 Augmenter la cohérence du système pour le rendre plus sûr

Au delà du besoin d'un contrôle en ligne afin d'augmenter la fiabilité des systèmes autonomes, le travail effectué au cours de cette thèse nous a emmené à une réflexion plus poussée sur la conception des architectures qui offre des perspectives sur le long terme. Le R^2C tel qu'il est présenté ici a été pensé et conçu afin de s'adapter dans une architecture déjà existante. Le travail d'intégration de ce composant a donc porté principalement à adapter cet outil aux composants déjà existants. Il était bien sûr inconcevable de réinventer la roue et cette approche nous a permis de venir aux conclusions présentées ici dans un délai raisonnable.

Toutefois, les difficultés de modélisation rencontrées ainsi que celle de spécifier les contraintes de bon fonctionnement du système sont principalement liées à un certain manque d'unité des modèles et spécifications des divers outils utilisés pour concevoir une plate-forme de ce type. Cette idée va dans le sens des réflexions qui ont mené au développement de l'architecture IDEA. Si on s'intéresse par



(a) Domaine de la contrainte (b) CDD (c) OCRD

FIG. VI.2 – Contrainte plus fine entre 2 horloges

exemple aux difficultés d'intégration et spécification du R^2C à la lumière de cette assertion, on entrevoit des débuts de solution.

Du côté fonctionnel, il pourrait être intéressant de reporter le travail de modélisation et spécification des contraintes des services lors de leur conception et éviter de reporter ceci lors de l'intégration. La modélisation d'une activité associée à un service pourrait être ainsi plus fine et adaptée à son fonctionnement. On peut d'ailleurs constater que ceci a déjà été proposé comme extension de l'outil GenoM [Mallet 02]. De même on trouve un travail de modélisation plus poussé du fonctionnement et des interactions des composants dans [Morisset 04] afin d'automatiser la spécification des modalités d'actions gérées par le superviseur. Ces informations comme celles sur les contraintes de fiabilités exploitées par ExoGen, pourraient être attachées aux modules et ainsi spécifiées par le concepteur de ce dernier. Si les modèles sont clairement définis et spécifiés, ils pourraient être unifiés afin d'offrir une base unique pour les divers composants qui agissent sur le composant concerné. Cela évite ainsi de multiplier les représentations d'une information unique et diminue les risques de fautes de conception.

Du côté décisionnel, nous avons pu voir tout au long de ce travail et, plus

particulièrement dans le chapitre IV, que la difficulté se trouve dans le fait que notre contrôleur va agir à l'encontre des décisions prises à ce niveau. Nous avons commencé à proposer des solutions limitant les conséquences des actions du R^2C, mais ceci peut encore être amélioré. Pour ce faire, il faut une meilleure harmonie entre notre contrôleur et les composants de plus haut niveau. Cette harmonie ne doit pas se limiter à des solutions ad hoc, mais il faut effectuer un travail de réflexion sur la formalisation de cette dernière. Grâce à cette formalisation on pourrait arriver à une meilleure communication entre ces deux composants qui permettrait ainsi de limiter les conflits d'intérêt entre ceux-ci tout en améliorant la fiabilité globale du système.

Les deux propositions données ci-dessus mettent en avant un réel besoin d'améliorer et formaliser les interfaces entre les divers composants présents dans les architectures hiérarchisées. On peut constater que cette tendance devient de plus en plus forte dans la communauté robotique. L'architecture IDEA avec son unification totale des modèles de conceptions en est un représentant mais ce n'est pas le seul. On peut aussi se référer à [Bennett 05], où les auteurs présentent un système où planification et exécution partagent un même modèle permettant de contrôler la cohérence entre le plan et son exécution. Cette formalisation peut se faire suivant deux approches :

– l'une où l'on cherche à unifier les modèles exploités par les divers composants. Cette unification peut être explicite – comme dans IDEA – ou issue d'une traduction d'un formalisme de haut niveau vers un autre plus générique et simple. L'essentiel est que les divers composants exploitent un modèle sous-jacent qui soit identique. On pourrait ainsi offrir un meilleur contrôle sur la cohérence de l'ensemble.

– l'autre approche serait de chercher à formaliser les interfaces entre les composants. Une telle formalisation mettrait en évidence les mécanismes de traduction et les interactions entre les divers éléments de l'architecture. Cette formalisation permettrait d'automatiser ce travail sur des bases formelles et de limiter les risques de faute à ce niveau.

Cette réflexion ne remet pas en cause la nécessité d'un contrôle d'exécution assurant la fiabilité du système. Elle permettrait toutefois que celui-ci soit mieux intégré et que sa spécification ainsi que son action présentent moins de risques de fautes. Nous aurions ainsi un système avec une grande autonomie décisionnelle tout en offrant des garanties fortes sur sa sûreté de fonctionnement.

Références bibliographiques

[Akers 78] S. B. Akers. *Binary Decision Diagrams*. IEEE Transactions on Computers, vol. C-27, no. 6, 1978.

[Alami 98] R. Alami, R. Chatila, S. Fleury, M. Ghallab & F. Ingrand. *An Architecture for Autonomy*. International Journal of Robotics Research, Special Issue on Integrated Architectures for Robot Control and Programming, vol. 17, no. 4, pages 315–337, Avril 1998.

[Alami 00] R. Alami, R. Chatila, S. Fleury, M. Herrb, F. Ingrand, M. Khatib, B. Morisset, P. Moutarlier & T. Siméon. *Around the Lab in 40 days...* Dans IEEE ICRA, 2000.

[Alami 02] R. Alami, R. Chatila, F. Ingrand & F. Py. *Dependability Issues in a Robot Control Architecture*. Dans 2nd IARP/IEEE-RAS Joint Workshop on Technical Challenge for Dependable Robot in Human Environment, LAAS-CNRS, Toulouse, France, `http ://www.laas.fr/~felix/download.php/DRHE-ws02.pdf`.

[Aloul 03] F.A. Aloul, I.L. Markov & K.A. Sakallah. *FORCE : A Fast and Easy-To-Implement Variable-Ordering Heuristic*. Dans Great Lakes Symposium on VLSI (GLSVLSI), pages 116–119, 2003.

[Altisen 03] K. Altisen, A. Clodic, F. Maraninchi & E. Rutten. *Using Controller Synthesis to Build Property-Enforcing Layers*. Dans European Symposium on Programming (ESOP), Avril 2003.

[Alur 93] R. Alur, C. Courcoubetis & D. Dill. *Model-Checking in Dense Real-time*. Information and Computation, vol. 1, no. 104, pages 2–34, 1993.

[Alur 94] R. Alur & D. Dill. *A theory of timed automata*. Theoritical Computer Science, vol. 2, no. 126, pages 183–235, 1994.

[Avizienis 04] A. Avizienis, J.C. Laprie & B. Randell. *Dependability and its threats : a taxonomy*. Dans 18th IFIP World Congress. Building the Information Society, volume 12, pages 91–120, Toulouse, Juin 2004. Kluwer Academic Publishers.

[Bengtsson 95] J. Bengtsson, K. G. Larsen, F. Larsson, P. Pettersson & W. Yi. UPPAAL — *a Tool Suite for Automatic Verification of Real–Time Systems*. Dans Proc. of Workshop on Verification and Control of Hybrid Systems III, numéro 1066 in Lecture Notes in Computer Science, pages 232–243. Springer–Verlag, Octobre 1995.

[Bennett 05] M. B. Bennett, R. L. Knight, R. D. Rasmussen & M. D. Ingham. *State-Based Models for Planning and Execution*. Dans workshop on Plan execution (ICAPS), 2005.

[Bensalem 05] S. Bensalem, M. Bozga, M. Krichen & S. Tripakis. *Testing conformance of real-time applications : Case of Planetary Rover*. Dans Verification and Validation meets Planning and Scheduling, Monterey (CA), Avril 2005.

[Bernard 00] D. Bernard, G. A. Dorais, E. Gamble, B. Kanefsky, J. Kurien, W. Millar, N. Muscettola, P. Nayak, N. Rouquette, K. Rajan, B. Smith, W. Taylor & Y. W. Tung. *Remote Agent Experiment*. Rapport technique, Nasa ARC et JPL, Février 2000.

[Berthomieu 03] B. Berthomieu & F. Vernadat. *State class constructions for branching analysis of Time Petri nets*. Dans Proceedings of TACAS 2003, page 442, 2003.

[Boussinot 91] F. Boussinot & R. de Simone. *The ESTEREL Language*. Proceeding of the IEEE, pages 1293–1304, Septembre 1991.

[Brace 91] K.S. Brace, R.L. Rudell & R.E. Bryant. *Efficient Implementation of a BDD Package*. Dans 27th ACM/IEEE Design Automation Conference, pages 40–45, 1991.

[Brooks 86] R.A. Brooks. *A Robust Layered Control System for a Mobile Robot*. IEEE journal of Robotics and Automation, vol. RA-2, no. 1, pages 14–23, Mars 1986.

[Brooks 91] R.A. Brooks. *Intelligence without Reason*. Dans International Joint Conference on Artificial Intelligence, volume 1, pages 569–595, Sydney, Aout 1991.

[Bryant 86] R. E. Bryant. *Graph-Based Algorithms for Boolean Function Manipulation*. IEEE Transactions

on Computers, vol. C-35, no. 8, pages 677–691, `citeseer.nj.nec.com/bryant86graphbased.html`.

[Bualat 04] M.G. Bualat, G.C. Kuntz, A.R. Wright & I.A. Nesnas. *Developing an Autonomy Infusion Infrastructure for Robotic Exploration*. Dans Proceedings of the 2004 IEEE Aerospace Conference, Big Sky, Montana, Mars 2004.

[Burch 92] J.R. Burch, E.M. Clarke, K.L. McMillan, D.L. Dill & L.J. Hwang. *Symbolic Model Checking : 10^{20} States and Beyond*. Information and Computing, vol. 98, no. 2, pages 142–170, `citeseer.nj.nec.com/burch92symbolic.html`.

[Bénazéra 03] E. Bénazéra. *Diagnostic et reconfiguration basés sur des modèles hybrides concurrents. Application aux satellites autonomes*. PhD thesis, Université Paul Sabatier de Toulouse, 2003.

[Corbett 00] J.C. Corbett, M. B. Dwyer, J. Hatcliff, S. Laubach, C. S. Păsăreanu, Robby & H. Zheng. *Bandera : extracting finite-state models from Java source code*. Dans International Conference on Software Engineering, pages 439–448, `citeseer.ist.psu.edu/corbett00bandera.html`.

[Coudert 93] O. Coudert, J. Madre, H. Fraisse & H. Touati. *Implicit Prime Cover Computation : An Overview*. Dans SASIMI '93, pages 413–422, Nara, Japan, Octobre 1993.

[de Medeiros 96] A. D. de Medeiros, R. Chatilla & S. Fleury. *Specification and Validation of a Control Architecture for Autonomous Mobile Robots*. Dans IROS, pages 162–169. IEEE, 1996.

[de Moura 04] L. de Moura, S. Owre, H. Rueß, J. Rushby, N. Shankar, M. Sorea & A. Tiwari. *SAL 2*. Dans Rajeev Alur & Doron Peled, editeurs, Computer-Aided Verification, CAV 2004, volume 3114 of *Lecture Notes in Computer Science*, pages 496–500, Boston, MA, Juillet 2004. Springer-Verlag.

[Emerson 90] E. A. Emerson. Temporal and modal logic. Handbook of Theoritical Computer Science, Elsevier, 1990.

[Espiau 95] B. Espiau, K. Kapellos & M. Jourdan. *Formal verification in robotics : Why and how*. Dans The International Foundation for Robotics Research, editor, The Seventh International Symposium of Robotics Research, pages 201 – 213, Munich, Germany, Octobre 1995. Cambridge Press.

[Fikes 72] R. Fikes, P. Hart & N. Nilsson. *Learning and Executing Generalized Robot Plans*. Artificial Intelligence, 1972.

[Fleury 94] S. Fleury, M. Herrb & R. Chatila. *Design of a Modular Architecture for Autonomous Robot*. Dans IEEE International Conference on Robotics and Automation, Atlanta, USA, 1994.

[Fleury 96] S. Fleury. *Architecture de Contrôle Distribuée pour Robots Mobiles Autonomes : Principes, Conception et Applications*. PhD thesis, Ecole Nationale Supérieure d'Arts et Métiers, 1996.

[Gat 97] E. Gat. *On Three-Layer Architectures*. Artificial Intelligence and Mobile Robots, 1997.

[Ghallab 88] M. Ghallab & H. Philippe. *A Compiler for Real-Time Knowledge-Based Systems*. Dans IEEE International Workshop on Artificial Intelligence for Industrial, Hitachy City, Japan, Mai 1988.

[Ghallab 01] M. Ghallab, F. Ingrand, S. Lemai & F. Py. *Architecture and Tools for Autonomy in Space*. Dans ISAIRAS, Montreal, Canada, `http ://www.laas.fr/~felix/download.php/isairas01.pdf`.

[Ghallab 04] M. Ghallab, D. Nau & P. Traverso. Automated planning theory and practice. Elsevier, 2004.

[Havelund 00] K. Havelund & T. Pressburger. *Model Checking Java Programs using Java PathFinder*. International Journal on Software Tools for Technology Transfer (STTT), vol. 2, no. 4, Avril 2000.

[Hayes-Roth 85] B. Hayes-Roth. *A Blackboard Architecture for Control*. Artificial Intelligence, vol. 26, no. 3, pages 251–321, Juillet 1985.

[Hayet 02] J.B. Hayet, F. Lerasle & M. Devy. *A visual landmark framework for indoor mobile robot navigation*. Dans ICRA, Washington DC (USA), Mai 2002.

[Holzman 97] G. J. Holzman. *The Model Checker Spin*. IEEE Trans. on Software Engineering, vol. 23, no. 5, pages 279–295, Mai 1997.

[Hu 93] Alan J. Hu & David L. Dill. *Reducing BDD Size by Exploiting Functional Dependencies*. Dans Design Automation Conference, pages 266–271, `citeseer.ist.psu.edu/hu93reducing.html`.

[Ingrand 96] F.F. Ingrand, R. Chatila, R. Alami & F. Robert. *PRS : A High Level Supervision and Control Language for Autonomous Mobile Robots*. Dans IEEE International Conference on Robotics and Automation, Mineapolis, USA, 1996.

[Jung 04] I. K. Jung. *Simultaneous localization and mapping in 3D environments with stereovision*. PhD thesis, Institut National Polytechnique de Toulouse, Mars 2004.

[Laborie 96] P. Laborie. *IxTeT un système de planification hiérarchique gérant le temps et les ressources*. Dans RFIA, pages 870–878, Rennes, 1996.

[Larsen 99] Kim G. Larsen, Justin Pearson, Carsten Weise & Wang Yi. *Clock Difference Diagrams*. Nordic Journal of Computing, vol. 6, no. 3, pages 271–??, `citeseer.ist.psu.edu/article/larsen99clock.html`.

[Lemai 04] S. Lemai. *IXTET-EXEC : planning, plan repair and execution control with time and resource management*. PhD thesis, Institut National Polytechnique, Toulouse, Juin 2004.

[Long 03] D. Long & M. Fox. *PDDL 2.1 : An Extension to PDDL for Expressing Temporal Planning Domains*. Artificial Intelligence, vol. 20, no. 20, pages 61–124, 2003.

[Lussier 04] B. Lussier, R. Chatila, F. Ingrand, M.O. Killijian & D. Powell. *On Fault Tolerance and Robustness in Autonomous Systems*. Dans 4th IARP/IEEE-RAS Joint Workshop on Technical Challenge for Dependable Robots in Human Environments, Manchester, Septembre 2004.

[Mallet 01] A. Mallet. *Localisation d'un robot mobile autonome en environnements naturels*. PhD thesis, Institut National Polytechnique de Toulouse, Juillet 2001.

[Mallet 02] A. Mallet, S. Fleury & H. Bruyninckx. *A specification of generic robotics software components : future evolution of GenoM in the Orocos context*. Dans International Conference on Intelligent Robots and Systems (IROS), Lausanne (CH), Octobre 2002.

[Marchand 00] H. Marchand, P. Bournai, M. Le Borgne & P. Le Guernic. *Synthesis of Discrete-Event Controllers based on the Signal Environment*. Discrete Event Dynamical System : Theory and Applications, vol. 10, no. 4, pages 325–346, Octobre 2000.

[McCluskey 56] E.J. McCluskey. *Algebraic Minimization and the design of two-terminal contact networks*. Bell System Technical Journal, vol. 35, pages 1417–1444, 1956.

[McMillan 93] K. L. McMillan. *Symbolic Model Checking*. PhD thesis, Kluwer Academic Publ., 1993.

[Minguez 04] J. Minguez, J. Osuna & L. Montano. *A "Divide and Conquer" Strategy based on Situations to achieve Reactive Collision Avoidance in Troublesome Scenarios*. Dans ICRA, 2004.

[Morisset 02] B. Morisset. *Vers un robot au comportement robuste. Apprendre à combiner des modalités sensori-motrices complémentaires.* PhD thesis, Université Paul Sabatier, Toulouse, Novembre 2002.

[Morisset 04] B. Morisset, G. Infantes, M. Ghallab & F. Ingrand. *Robel : Synthesizing and Controlling Complex Robust Robot Behaviors*. Dans 4th International Cognitive Robotics Workshop (CogRob), Aout 2004.

[Moutarlier 91] P. Moutarlier & R. Chatila. *Incremental Free-Space Modeling from Uncertain Data by an Autonomous Mobile Robot*. Dans Geometric Reasoning for Perception and Action, pages 200–213, Septembre 1991.

[Muscettola 98] N. Muscettola, P. P. Nayak, B. Pell & B. Williams. *Remote Agent : To Boldly Go Where No AI System Has Gone Before*. Artificial Intelligence, vol. 103, 1998.

[Muscettola 02] N. Muscettola, G. A. Dorais, C. Fry, R. Levinson & C. Plaunt. *IDEA : Planning at the Core of Autonomous Reactive Agents*. Dans Proceedings of the 3rd International NASA Workshop on Planning and Scheduling for Space, Octobre 2002.

[Musliner 93] D. J. Musliner, E. H. Durfee & K. G. Shin. *CIRCA : A Cooperative Intelligent Real Time Control Architecture*. IEEE Transactions on Systems, Man, and Cybernetics, vol. 23, no. 6, pages 1561–1574, `citeseer.nj.nec.com/musliner93circa.html`.

[Nesnas 03] I.A. Nesnas, A. Wright, M. Bajracharya, R. Simmons & T. Estlin. *CLARAty and Challenges of Developing Interoperable Robotic Software*. Dans International Conference on Intelligent Robots and Systems (IROS), Nevada, Octobre 2003. invited paper.

[Pagetti 04] Claire Pagetti. *Extension temps-réel d'AltaRica*. PhD thesis, IRCCyN, Ecole centrale de Nantes and Université de Nantes, Avril 2004.

[Py 02a] F. Py & F. Ingrand. *An Execution Control System for Autonomous Robots*. Dans IEEE International Conference on Robotics and Automation, Washington DC (USA), Mai 2002.

[Py 02b] F. Py & F. Ingrand. *Online Execution Control Checking for Autonomous Systems*. Dans International Conference on Intelligent Autonomous Systems, Marina del Rey USA, Mars 2002.

[Py 04a] F. Py & F. Ingrand. *Dependable Execution Control for Autonomous Robots*. Dans International Conference on Intelligent Robots and Systems, Sendai, Japon, `http ://www.laas.fr/~felix/download.php/iros04-r2c.pdf`.

[Py 04b] F. Py & F. Ingrand. *Real-Time Execution Control for Autonomous Systems*. Dans Embedded Real Time Systems, Toulouse, France, `http ://www.laas.fr/~felix/download.php/erts04-1.pdf`.

[Rémond 01] Yann Rémond. *Un support langage pour les modes de fonctionnement des systèmes temps-réel : extension de LUSTRE par des automates de modes*. PhD thesis, Université de GRENOBLE I, Octobre 2001.

[Rudell 93] R. Rudell. *Dynamic Variable Ordering for Ordered Binary Decision Diagrams*. Dans IEEE/ACM ICCAD'93, pages 42–47, 1993.

[Scherer 05] S. Scherer, F. Lerda & E. M. Clarke. *Model Checking of Robotic Control Systems*. Dans iSAIRAS, Munich (DE), 2005.

[Shnoebelen 99] P. Shnoebelen & al. Vérification de logiciels - techniques et outils du model-checking. Numeéro ISBN 2-7117-8646-3. Vuibert Informatique, Mai 1999.

[Simmons 94] R. Simmons. *Structured Control for Autonomous Robots*. IEEE Transactions on Robotics and Automation, vol. 10, no. 1, pages 31–43, Février 1994.

[Thomas 90] W. Thomas. Automata on infinite objects, volume B, chapitre 4, pages 133–191. Elsevier Sciences Publishers, 1990.

[Tigli 93] Jean-Yves Tigli, Michel Occello & M.-C. Thomas. *Toward a New Intelligent Reactive Controller for Autonomous Mobile Robots*. Dans ICRA (3), pages 249–254, 1993.

[Trinquart 02] R. Trinquart, S. Lemai & S. Cambon. *One step on the left, one step on the right and back to the middle : exploring temporal domains in a POP fashion*. Dans Workshop on Planning for Temporal Domains, pages 41–48, Toulouse (FR), Avril 2002.

[Tripakis 02] S. Tripakis. *Fault diagnosis for timed automata*. Dans Formal Techniques in Real Time and Fault Tolerant Systems (FTRTFT), printemps 2002.

[Volpe 01] R. Volpe, I.A.D. Nesnas, T. Estlin, D. Mutz, R. Petras & H. Das. *The CLARAty Architecture for Robotic Autonomy*. Dans Proceedings of the 2001 IEEE Aerospace Conference, Big Sky Montana, Mars 2001.

[Williams 03a] B. C. Williams, M. Ingham, S. H. Chung & P. H. Elliot. *Model-based Programming of Intelligent Embedded Systems and Robotic Space Explorers*. Proc. of the IEEE : Special Issue on Modeling and Design of Embedded Software, vol. 9, no. 1, pages 212–237, Janvier 2003.

[Williams 03b] B. C. Williams, M. D. Ingham, S. Chung, P. Elliott, M. Hofbaur & G. T. Sullivan. *Model-Based Programming of Fault-Aware Systems*. Artificial Intelligence, pages 61–75, hiver 2003.

Annexe A

Règles ExoGen utilisées pour dala.

Voici l'ensemble des règles utilisées sur le robot Dala tel que décrit dans le chapitre V. Pour des raisons de clarté nous n'avons pas mis ici les déclarations des différents services de la couche fonctionnelle et chaque règle est précédée d'un commentaire donnant brièvement la contrainte du système à laquelle elle correspond.

```
#define refME "rflex"

check {
 /*
  * On ne peut attacher de Motion Estimator (ME) ou de Sensor
  * Estimator (SE) à POM que si le modèle géométrique du robot a déjà
  * été défini
  */
 always: ( running(pom.addME) || running(pom.addSE) )
        => last(pom.SetModel);

 /*
  * Le seul ME de référence possible est "rflex"
  */
```

```
never: running(pom.SetRefME with arg.name!=refME);


/*
 * Pour pouvoir définir le ME de référence de POM il faut que ce
 * dernier ait déjà été attaché. A cause des limitations actuelles du
 * langage on va contraindre que le ME de référence doit être le
 * dernier ajouté (il faudra corriger ceci soit en donnant la
 * possibilité au R2C d'exploiter des variables internes, soit en
 * étendant son support de la CTL).
 *
 */
always: running(pom.setRefME) => last(pom.addME with arg.name==refME);

/*
 * pom ne peut être lancé que si pom.setRefME a été exécutée correctement.
 */
always: running(pom.Run) => last(pom.setRefME);


/*
 * on ne peut naviguer avec ndd que si pom est lancé
 *
 * NB : Après avoir demandé aux personnes qui ont développé pom
 * on peut considérer que pom a été lancé une fois que
 * pom.Run a été appelé et que les règles plus haut ont été
 * respectées
 *
 * Après utilisation dans la démo on pourrait expliciter ceci sur
 * dala en vérifiant que les ME qui nous intéressent ont été ajoutés
 */
always: running(ndd.GoTo) => last(pom.Run);

/*
 * ndd ne peut être utilisé si la requête d'initialisation setParams
 * n'a pas été appelée et si la vitesse lineaire max n'a pas été
 * fixée à moins de 1m/s
 */
always: running(ndd.GoTo) => ( last(ndd.SetParams)
```

```
                                && last(ndd.SetSpeed with
                                          arg.linear<1.0 ) );

/*
 * ndd.Goto utilise le poster produit par aspect.AspectFromPosterConfig
 * il est donc nécessaire que cette requête s'exécute en même temps que
 * ndd.Goto
 */
always: running(ndd.GoTo) => running(aspect.AspectFromPosterConfig);

/*
 * aspect.AspectFromPosterConfig ne peut tourner que si aspect a été
 * initialisé
 */
always: running(aspect.AspectFromPosterConfig) =>
          last(aspect.SetViewParameters);

/*
 * si le poster que lit aspect.AspectFromPosterConfig est
 * pomSickFramePos il faut que sick ait été attaché à POM
 */
always: running(aspect.AspectFromPosterConfig with
                arg.posPosterName.name.name=="pomSickFramePos")
          => last(sick.SetPomTagging with arg==SICK_TRUE);

/*
 * si le poster que tracke aspect.AspectFromPosterConfig est
 * pomSickFramePos alors il faut que le producteur de ce poster
 * (sick.ContinuousShot) ait ete activé
 *
 * NB : la production de pomSickFramePos est lancé par une requête de
 * contrôle (ContinuousShot) et est actif une fois que cette requête
 * s'est terminée
 */
always: running(aspect.AspectFromPosterConfig with
                arg.posPosterName.name.name=="pomSickFramePos")
          => last(sick.ContinuousShot);

/*
```

```
 * On ne peut communiquer si le robot bouge
 *
 * NB: cette contrainte est directement liée a la manip et est
 * normalement aussi gérée par IxTeT. Le module antenna est d'ailleur
 * un module qui ne fait rien
 */
never: running(antenna.Comunicate) && running(rflex.TrackSpeedStart);

/*
 * Quand le robot bouge la platine ne doit pas bouger
 *
 * NB: cette contrainte est normalement plus complexe :
 *  "La platine ne doit pas bouger si rflex tracke un poster produit
 * à partir de donnés issus de la stéréo vision (P3D)."
 * Mais pour des raisons de non maintenance du code les modules
 * liés a p3d ne sont pas encore a jour et comme c'est une des
 * injections les plus "visuelles"  on a gardé la règle en la rendant
 * plus contraignante
 */
never: running(rflex.TrackSpeedStart) &&
         (
           running(platine.CmdPosCoord)
           || running(platine.CmdPosPan)
           || running(platine.CmdPosTilt)
           || running(platine.TrackPos)
         );

/*
 * le robot ne doit pas dépasser la vitesse de 0.9 m/s
 */
never: running(rflex.TrackSpeedStart with arg.name.value.v>0.9);

/*
 * avant de pouvoir communiquer il faut que le module anntenna ait
 * une connaissance des fenêtres de visibilité
 */
always: running(antenna.Comunicate) => last(antenna.AddWindow);

/*
```

```
  * Avant d'indiquer les fenêtres de visibilité a antenna ce module
  * doit avoir été initialisé.
  */
 always: running(antenna.AddWindow) => last(antenna.Init);

 /*
  * On ne peut prendre d'image si camera n'a pas été
  * initialisée avec succès
  */
 always: running(camera.OneShot) => last(camera.Initialize);

 /*
  * Lors d'une prise de vue la platine ne doit pas bouger.
  */
 never: running(camera.OneShot) &&
          (
            running(platine.CmdPosCoord)
            || running(platine.CmdPosPan)
            || running(platine.CmdPosTilt)
            || running(platine.TrackPos)
          );

}
```

Résumé Il y a un besoin grandissant d'autonomie dans des systèmes temps-réel complexes tels que les robots ou les satellites. Ceci met en avant un problème non trivial : d'un côté il y a des systèmes complexes - donc difficiles à valider - avec une intervention de l'humain dans la boucle qui se veut minimale, de l'autre nous avons des domaines où il faut que le système ait un comportement sûr fonctionnellement afin d'éviter les coûts financiers et/ou humains d'une perte ou d'une détérioration du système. Ces deux notions mises en vis-à-vis semblent contradictoires. En effet comment être sûr qu'un système autonome avec un pouvoir décisionnel fort, n'aura pas un comportement non nominal qui pourra menacer le bon déroulement de la mission ? Comment être sûr qu'un satellite n'allumera pas ses réacteurs sans avoir protégé ses capteurs fragiles (objectif de la caméra, ...) ? Une réponse partielle à ce type de problèmes pourrait être d'utiliser un planificateur de haut niveau qui ne donnerait que des plans garantis comme sûrs et valides. Toutefois ces planificateurs n'ont pas un modèle complet des actions qu'ils effectuent et de leurs conséquences. En effet, les direc-tives données par ce planificateur sont généralement affinées en sous tâches par un superviseur. Le planificateur n'a donc pas un contrôle complet sur le moyen d'effectuer cette action. Nous présentons ici les travaux effectués afin d'intégrer un système de contrôle d'exécution en ligne dans une architecture hiérarchisée. Nous décrivons ici la nécessité et le rôle d'un tel compo-sant dans ce type d'architecture. Nous introduisons le R2C, notre contrôleur basé sur les hypothèses synchrones, ainsi que l'outil permettant sa génération en exploitant des techniques issues du model-checking symbolique. Enfin nous discutons de la nécessité de prendre en compte les composants décisionnels dans le contrôle afin d'interférer le moins possible avec les décisions prises par ceux-ci. Les résultats obtenus durant des expérimentations sur une plate-forme robotique confirment les idées développées au cours de ce travail et permettent d'en tirer les conclusions et perspectives sur la mise en place d'un contrôle pour l'amélioration de la fiabilité globale de tels systèmes.

Abstract There is an increasing need for advanced autonomy in complex embedded real-time systems such as robots, satellites, or UAVs. Still, this raises a major problem : on one side we have complex sys-tems - therefore, hard to validate - with little human intervention, on the other side these systems are used in domains where safety is critical. These two concerns seem to be contradictory. Indeed, how can we guaranty that an autonomous system, with high level decisional capabilities, will exhi-bit a proper behavior and will not jeopardize the mission ? How can we be sure that the satellite jets are not fired while fragile sensors are left unprotected (camera lens, ...) ? An answer to this problem may be to implement a high level planner that would only produce safe and valid plans. However, these planners do not have a complete model of all atomic actions and their consequences. Indeed tasks produced by planner are refined into executable tasks by a su-pervisor. As a consequence the planner do not have a full control on how its tasks are being exe-cuted. The work we present here integrate an on-line execution control component for hierarchical archi-tectures. We first describe the role of this program in this kind of architecture. Then we introduce the R2C, our controller based on synchronous hypothesis, and the tool used to generate this controller using symbolic model-checking techniques. We then discuss why it is important to take into account the decisional components in our controller to minimize the controller counter effects. We eventually illustrate our contribution with some experimental results produced and gathered on our robotics platforms. We then conclude and give some possible future work in this area.