Validation d'architectures temps-réel pour la robotique autonome

Nicolas Gobillot

THÈSE

En vue de l'obtention du

DOCTORAT DE L'UNIVERSITÉ DE TOULOUSE

Délivré par :

Institut Supérieur de l'Aéronautique et de l'Espace

Présentée et soutenue par :
Nicolas GOBILLOT

le vendredi 29 avril 2016

Titre :

Validation d'architectures temps-réel pour la robotique autonome

---

École doctorale et discipline ou spécialité :

EDSYS : Robotique et Informatique

Unité de recherche :

Équipe d'accueil ISAE-ONERA CSDV

Directeur(s) de Thèse :

M. Charles LESIRE-CABANIOLS (directeur de thèse)
M. David DOOSE (co-directeur de thèse)

Jury :

M. Frédéric BONIOL, Professeur d'Université - Président du jury
M. David ANDREU, Maître de conférences - Rapporteur
M. Saddek BENSALEM, Professeur d'Université
M. Noury BOURAQADI, Professeur Ecole des Mines Douai
M. David DOOSE, Ingénieur de recherche ONERA - Co-directeur de thèse
M. Simon LACROIX, Directeur de recherche CNRS
M. Charles LESIRE-CABANIOLS - Ingénieur de recherche ONERA - Directeur de thèse
M. Nicolas NAVET, Maître de conférences - Rapporteur

Remerciements

Toute ma gratitude va à Messieurs Charles Lesire et David Doose, mes deux Directeurs de thèse. Tout d'abord, ils m'ont accueilli dans leur unité de recherche à l'ONERA de Toulouse. Ensuite ils m'ont guidé pendant ces trois années en me prodiguant leurs conseils. Ainsi j'ai pu développer les compétences et la rigueur nécessaire à l'aboutissement de ce travail. Je les remercie pour tout ceci et les assure de toute ma reconnaissance.

Table des matières

Table des figures

Liste des tableaux

Liste des algorithmes

Liste des listings

Table des sigles

τ	Tâche
T	Période d'exécution
P	Priorité
D	Échéance
R	Date de réveil
C	Temps d'exécution
$\mathcal{C}$	Pire temps d'exécution
$\mathcal{R}$	Pire temps de réponse
$\mathcal{X}$	Temps d'exécution mesuré
$\overline{\mathcal{X}}$	Temps d'exécution probabiliste
SM	Machine à état
s	État
S	Ensemble d'états
e	Transition de machine à état
E	Ensemble de transition de machine a état
σ	Transition de machine à état périodique
Σ	Ensemble de transition de machine à état périodique
δ	Délai d'une transition
Δ	Fonction de délai
$\mathcal{T}$	Trace
$\mathcal{U}$	Ensemble de traces
$\mathcal{V}$	Ensemble de traces optimisé
ϕ	Trace faisable
rbf	Fonction de requête

Introduction

Les robots ont dès leur création été utilisés dans des chaînes automatisées. L'environnement de ces robots est connu à priori et n'évolue pas dans le temps. Ils n'ont donc pas besoin de capacités de décision et suivent des programmes décidés à l'avance par un opérateur humain. L'efficacité de ces machines leur permet d'effectuer des tâches répétitives et/ou dangereuses comme dans des chaînes d'assemblage automobile.

La capacité des robots à travailler dans des environnements hostiles a permis leur utilisation dans de nombreux domaines. Ils ont par exemple trouvé des applications dans le domaine spatial, nucléaire ou militaire. L'augmentation des performances des calculateurs et des capteurs a permis aux robots de gagner en autonomie. Les robots peuvent désormais évoluer dans des environnements de plus en plus complexes et réaliser un certain nombre de tâches automatiquement, sans l'aide d'un opérateur humain.

Quelque soit le niveau d'autonomie de ces machines, il est primordial de garantir qu'aucune défaillance matérielle ou logicielle n'entraîne de catastrophe tant au niveau matériel que humain. Pour cela, il est nécessaire d'effectuer des analyses sur les différents constituants du robot. Un système robotique est constitué de trois éléments principaux : un système mécanique, un système électronique et un système logiciel. Les parties mécaniques du robot définissent ses capacités à interagir avec l'environnement en lui fournissant des moyens de locomotion, de perception et d'action. L'électronique du robot sert d'interface entre les parties mécaniques et logicielles tout en apportant d'autres moyens de perception et de proprioception. Dans le cadre de ces travaux de thèse, nous nous intéressons principalement à l'analyse de la partie logicielle.

La partie logicielle est, elle aussi, séparée en couches : le système d'exploitation et l'application ; un middleware pouvant être utilisé entre l'application et le système d'exploitation. Le système d'exploitation a pour rôle de gérer les éléments matériels de l'électronique du robot et d'allouer aux applications les ressources nécessaires à leur exécution. Le middleware, quant à lui, abstrait les spécificités d'implémentation du système d'exploitation afin de simplifier le développement des applications. Enfin, les applications définissent la fonctionnalité et les missions du robot.

Le logiciel d'un système robotique est donc un système complexe. Afin de simplifier la conception de ces machines, le développement est décomposé en modules qui sont ensuite assemblés pour constituer le système complet. Cependant, la facilité de conception de ces systèmes est bien souvent contrebalancée par la complexité de leur mise en sécurité, à la fois d'un point de vue comportemental et temporel.

Le nombre de modules impliqués au sein d'une application robotique augmente avec la

complexité des missions confiées aux robots ainsi qu'avec les performances des calculateurs embarqués. Selon la méthode d'implémentation de ces modules sur le système logiciel, le nombre de tâches associées à chaque module varie de une à quelques dizaines. Le middleware ROS [Quigley et al. 2009], par exemple, associe une tâche par module mais aussi une tâche par émetteur/récepteur de données lié au module. Le nombre de tâches impliquées dans un système logiciel augmente donc rapidement et les interactions entre ces tâches deviennent difficiles à prédire. Afin de maintenir un bon niveau de sécurité lors de l'exécution de tels systèmes, il faut analyser le comportement et les interactions de ces tâches.

Il existe des ensembles d'outils et de méthodes d'analyse d'ordonnancement d'un système logiciel à base de tâches. Ces outils permettent de vérifier la capacité d'ordonnancement d'un système de tâches sur un système matériel donné. Cependant ces méthodes d'analyse considèrent les tâches comme des entités monolithiques, sans prendre en compte la structure interne des tâches.

Problématique de l'étude

Cette étude consiste à prendre en compte la structure interne des tâches dans des méthodes d'analyse d'ordonnançabilité. Ceci dans le but de vérifier si le découpage de tâches monolithiques permet d'améliorer la précision des analyses d'ordonnancement. De plus, les méthodes d'analyse d'ordonnancement temps-réel sont à la fois complexes à mettre en œuvre et coûteuses en termes de temps de calcul. Cette étude consiste donc aussi à vérifier l'applicabilité de telles méthodes d'analyse à des cas d'applications robotiques.

Plan de lecture

Ce document est découpé en trois parties : la première décrit le modèle utilisé dans la seconde partie ; la seconde partie est consacrée à l'analyse temporelle ; enfin la troisième partie est dédiée à une expérimentation réelle sur un cas robotique.

Chapitre 1 : État de l'art

Ce chapitre présente les précédentes contributions sur plusieurs domaines. Ce chapitre est donc séparé en quatre parties : la première décrit quelques architectures robotiques existantes ; la seconde partie concerne les *middlewares* et leurs spécificités ; la troisième partie aborde les principaux langages de modélisation ; enfin la dernière partie est dédiée aux différentes analyses temps-réel existantes.

Chapitre 2 : Méthodologie

Ce chapitre propose un protocole à suivre afin de concevoir, modéliser et analyser une architecture logicielle robotique.

Partie I : Modèle de composant, d'architecture et de déploiement

Cette partie, séparée en trois chapitres, aborde le langage de modélisation Mauve, le modèle d'exécution associé ainsi que la méthode d'obtention des temps d'exécution des tâches.

Chapitre 3 : Langage de modélisation Mauve

Ce chapitre définit le langage de modélisation Mauve permettant de décrire des tâches logicielles sous forme de composants. Il permet, de plus, de modéliser des architectures afin d'instancier un ensemble de tâches. Enfin, Mauve modélise le déploiement d'architectures en définissant les paramètres d'exécution des tâches.

Chapitre 4 : Développement d'un modèle d'exécution de tâches

Ce chapitre définit un modèle d'exécution utilisant un modèle de tâches préemptibles à base de machines à états. Pour cela, nous avons développé un modèle de tâche capable d'exploiter les spécificités des machines à états dans le but de vérifier si la prise en compte de ces machines à états permet d'améliorer la précision des analyses d'ordonnancement.

De plus, ce chapitre présente les méthodes d'obtention des temps d'exécution des tâches logicielles ainsi que des fonctions contenues dans ces tâches.

Partie II : Analyse d'ordonnançabilité utilisant des machines à états

Cette seconde partie traite de la méthode d'analyse d'ordonnançabilité permettant d'exploiter le modèle de tâches à base de machines à états.

Chapitre 5 : Protocole d'analyse

Ce chapitre présente les différentes étapes du protocole d'analyse mis en place au chapitre 6.

Chapitre 6 : Conception d'une méthode d'analyse utilisant le modèle de tâches à base de machines à états

Ce chapitre développe des outils d'analyse temporelle utilisant les modèles de tâches, d'architecture et de déploiement. Pour cela, nous avons conçu une méthode d'analyse temporelle exploitant les machines à états des tâches.

Ce chapitre s'intéresse aussi à l'estimation de l'impact de l'utilisation des machines à états en termes de complexité d'analyse ainsi que de la précision du calcul du pire temps de réponse des tâches par rapport à une analyse ne les utilisant pas.

Partie III : Expérimentation

Cette troisième partie se concentre sur l'expérimentation du modèle de tâches ainsi que de son analyse sur un cas d'application robotique.

Chapitre 7 : Développement d'un logiciel de génération de code exécutable utilisant le modèle de tâches à base de machines à états

Ce chapitre explicite le développement d'un générateur de code exécutable. La génération de code exécutable permet à la fois de simplifier le développement d'architectures logicielles et de limiter les erreurs d'implémentation.

Chapitre 8 : Développement d'une architecture robotique modulaire

Ce chapitre propose le développement d'une architecture robotique d'exploration autonome. Pour cela, nous avons choisi d'utiliser un robot mobile ainsi qu'un calculateur embarqué et des capteurs. Nous avons conçu une architecture de type Navigation-Guidage-Contrôle afin de réaliser une mission d'exploration.

Chapitre 1

État de l'art

Sommaire

Une architecture robotique est constituée de deux éléments principaux : une partie matérielle et une partie logicielle. La partie matérielle définit les capacités intrinsèques du robot, comme sa capacité à se mouvoir grâce à ses actionneurs ou à effectuer des observations avec ses senseurs. La partie logicielle apporte, au matériel, les lois de commande et les capacités de décision nécessaires à l'accomplissement d'une mission.

Ces travaux de thèse s'articulent autour de la partie logicielle constitutive des systèmes robotiques. La complexité de mise en œuvre de tels systèmes à amené les développeurs à séparer la conception des logiciels robotiques afin d'en simplifier le développement ainsi que d'en analyser le comportement. Pour cela, les applications robotiques sont organisées

autour d'architectures à base de tâches afin d'en définir le comportement. Les architectures sont cependant complexes et nécessitent un niveau d'abstraction supplémentaire : la modélisation. La modélisation d'architectures permet d'abstraire l'implémentation des fonctionnalités et de se concentrer sur la structure et les interactions entre les tâches du système. De plus les langages de modélisation apportent le support d'outils d'analyse permettant de valider les comportements fonctionnels et temporels des architectures.

Dans ce chapitre, nous présentons donc quatre des éléments essentiels à la conception et à l'analyse d'architectures robotiques. Le premier élément est l'architecture robotique en elle même dont le rôle est de définir la structure logicielle embarquée au sein du robot. Le second élément correspond au *middleware* servant de support à l'architecture afin de permettre au développeur de se concentrer sur l'application en simplifiant son implémentation. L'élément suivant traite des langages de modélisation permettant de modéliser le fonctionnement de l'architecture. Le dernier élément est l'analyse temporelle permettant de vérifier, dans certaines limites, l'ordonnançabilité des architectures à partir des modèles.

1.1 Architectures logicielles de contrôle en robotiques

L'architecture robotique, aussi appelée architecture logicielle d'un robot correspond à son cœur logiciel. Son rôle est de définir le comportement global du robot. Il existe plusieurs types d'architectures robotiques : les structures hiérarchiques, comportementales ou hybrides. Dans tous les cas, il s'agit d'un assemblage de fonctions de plus ou moins haut niveau comme des algorithmes de contrôle du déplacement du robot, des filtres de données sur des observations de capteurs, de la cartographie, de la planification, etc.

1.1.1 Architectures hiérarchiques

Une architecture hiérarchique est une architecture robotique dont les instructions sont transmises principalement du haut de la hiérarchie vers le bas. La partie haut niveau de l'architecture contient un ensemble de fonctionnalités comportementales et décisionnelles définissant la mission du robot et se basant sur une modélisation du monde. Le bas niveau de l'architecture décrit les fonctions réactives du robot ainsi que les algorithmes de contrôle des actionneurs et des capteurs. Parmi les architectures hiérarchiques, nous pouvons citer *Three Layer Architecture (3T)* [Gat 1998], *4D-RCS* [Albus 2002], *Intelligent Distributed Execution Architecture (IDEA)* [Muscettola et al. 2002], *Mission Data System (MDS)* [Dvorak et al. 2000], *ORCCAD* [Borrelly et al. 1998], *RAS* [Georgas et Taylor 2007], *Saphira* [Konolige et Myers 1997] ou encore *Teleo-Reactive EXecutive (T-REX)* [C McGann et al. 2008]. La structure de ces architectures diffère, mais le

concept général est similaire ; à savoir, l'organisation hiérarchique des divers éléments constitutifs de l'architecture. A titre d'exemple, nous en détaillons deux : *Three Layer Architecture* et *4D/RCS*.

1.1.1.1 *Three Layer Architecture*

L'architecture *Three Layer Architecture* [Gat 1998] ou *3T* est une architecture robotique hiérarchique constituée de trois couches distinctes : le *contrôleur*, le *séquenceur* et le *délibérateur*. Ces trois couches sont organisées selon une structure hiérarchique : le contrôleur permet de gérer les interactions bas niveau avec le matériel ; le séquenceur sert d'interface entre le contrôleur et le délibérateur ; le délibérateur contient les algorithmes de planification haut niveau.

La couche bas niveau, le contrôleur, est constituée d'une ou plusieurs tâches implémentant une ou plusieurs boucles de contrôle. Celles-ci permettent de générer les comportements primitifs du robot en couplant des informations sensorielles à la génération de consignes d'actionnement par l'intermédiaire de fonctions de transfert. Lorsque plusieurs fonctions de transfert sont implémentées au sein de la même architecture, une seule est active à la fois.

Le séquenceur, quant à lui, sélectionne les comportements implémentés au niveau du contrôleur. La sélection d'un comportement s'effectue en fonction de l'environnement du robot et selon des conditions de continuité des fonctions de transfert. La fenêtre de calcul dédiée à cette couche de l'architecture ne doit pas être « trop longue » par rapport à l'évolution de l'environnement et au niveau d'abstraction apporté par le contrôleur.

Enfin, le délibérateur est dédié à la planification long terme, relativement au taux de variation de l'environnement. Il est constitué d'un ou plusieurs planificateurs selon les besoins de l'application. Il interagit directement avec le séquenceur selon deux méthodes : soit il fournit directement des plans à exécuter, soit il répond à des demandes du séquenceur.

1.1.1.2 4D-RCS

L'architecture *4D/RCS* [Albus 2002] est une architecture à base de composants destinée à être déployée sur des robots mobiles terrestres d'exploration. Elle consiste en un assemblage de nœuds logiciels hiérarchiques constitués de quatre éléments chacun. Ces nœuds sont structurellement identiques et contiennent des fonctionnalités de traitement de données capteur (SP), de modélisation de l'environnement (WM), de planification (VJ) et de génération comportementale (BG) comme le montre la figure 1.1. Selon la position

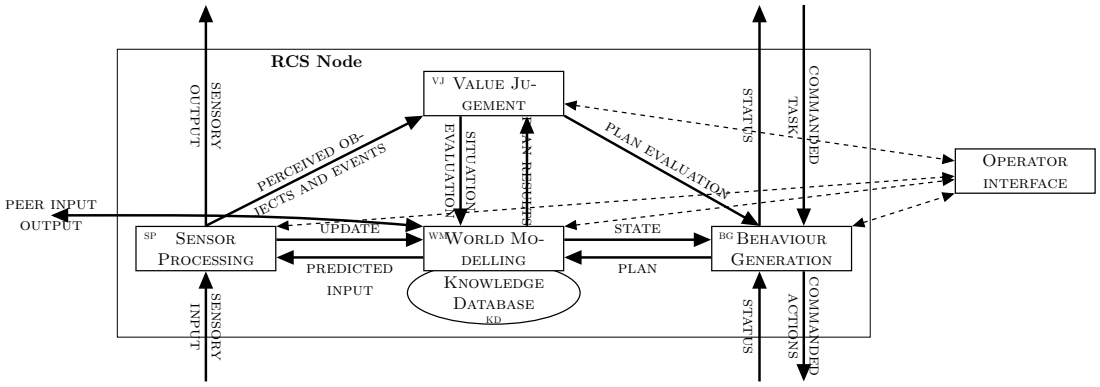


FIGURE 1.1 – Structure d'un nœud de l'architecture 4D/RCS

hiérarchique du nœud dans l'architecture, la résolution spatiale et temporelle des calculs varie : lorsque les nœuds sont en bas de la hiérarchie, ils possèdent une période faible et une résolution spatiale élevée afin de fournir des comportements bas-niveau réactifs. A contrario, un nœud de haut niveau va raisonner sur le long terme et sur de grandes distances pour produire des plans abstraits.

Les différents niveaux hiérarchiques sont reliés par l'intermédiaire de connexions entre nœuds : les blocs de génération comportementale (BG) sont connectés sous forme d'un arbre de commande et les blocs de traitement de données capteur (SP) suivent la hiérarchie sous la forme d'un graphe. De plus, pour chaque niveau hiérarchique, les nœuds partagent une base de donnée utilisée dans les blocs de modélisation de l'environnement (WM).

Pour chaque niveau hiérarchique, les horizons temporels et spatiaux sont différents. Ceci permet d'obtenir des raisonnements et des comportements complexes en un nombre réduit de niveaux. Les horizons temporels et spatiaux sont généralement augmentés d'un ordre de grandeur par niveau hiérarchique. Par exemple, le niveau le plus bas, à savoir le niveau de commande, va avoir un horizon temporel de 50 ms correspondant à un déplacement de 0.5 m si le robot se déplace à 10 m/s. La couche supérieure, quant à elle, planifie les mouvements du robot sur un horizon temporel de 500 ms pour un déplacement de 5 m. Ce processus permet de calculer des plans de plus en plus loin dans le temps et dans l'espace tout en conservant une bonne réactivité à court terme.

1.1.2 Architectures comportementales

Une architecture comportementale est l'opposée de l'architecture hiérarchique : au lieu de modéliser l'environnement et de planifier sur ce modèle, elle consiste en un ensemble de comportements couplés permettant de produire des actions coordonnées.

Les comportements complexes sont découpés en comportements élémentaires qui, une fois associés, produisent la fonctionnalité attendue. Cette technique est plus robuste aux aléas d'exécutions que les architectures hiérarchiques puisque si un comportement élémentaire échoue, un comportement global dégradé est maintenu et permet de continuer le déroulement de la mission. Parmi les architectures comportementales, nous pouvons citer *Robust Layered Control System (subsumption)* [Anagnostopoulos, Iliou et Giannoukos 2015].

1.1.2.1 *Robust Layered Control System*

L'architecture *Robust Layered Control System* [Anagnostopoulos, Iliou et Giannoukos 2015], plus communément appelée *subsumption*, est une architecture comportementale constituée d'un ensemble de *couches de compétences*. Chaque couche de compétence consiste en une fonctionnalité complète utilisant des informations provenant de capteurs afin de produire un comportement et d'actionner les actionneurs du robot. Les couches de compétences sont organisées de manière hiérarchique selon la criticité de leur rôle et peuvent hériter de certaines fonctionnalités des niveaux hiérarchiques inférieurs comme le montre la figure 1.2.

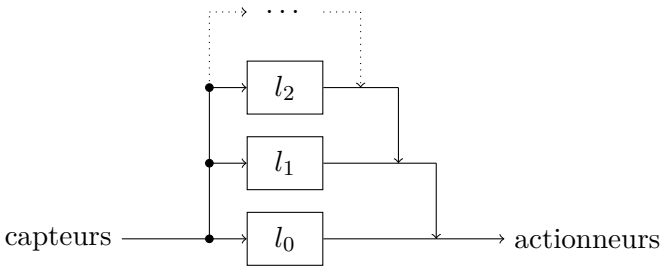


Figure 1.2 – Structure de contrôle de l'architecture *subsumption*

Autant de couches de compétence peuvent être ajoutées à cette architecture. Ces couches peuvent aller de simples algorithmes d'évitement d'obstacles à des capacités de planification, de cartographie ou de raisonnement sur le comportement d'objets dans l'environnement.

Les couches étant organisées selon un ordre de priorités, un mécanisme de contrôle de l'exécution des différentes couches est mis en place. Ce mécanisme donne la priorité aux niveaux hiérarchiques les plus hauts à la condition qu'ils aient terminé leur exécution dans le temps qui leur a été imparti ; si la couche de compétence de niveau n n'a pas terminé son exécution à temps, l'architecture donne l'exécution à la couche $n-1$. Ce mécanisme permet au robot de continuer son exécution dans un mode dégradé.

1.1.3 Architectures hybrides

Une architecture hybride est la combinaison des deux types d'architectures précédentes : la réactivité et la robustesse des architectures comportementales sont combinées aux capacités de décision et de planification des architectures hiérarchiques. Parmi les architectures hybrides, nous pouvons citer *Autonomous Robot Architecture (AuRA)* [Arkin et Balch 1997] ou *Coupled Layer Architecture for Robotic Autonomy (CLARAty)* [Volpe et al. 2001].

1.1.3.1 CLARAty

L'architecture *CLARAty* [Volpe et al. 2001] est une architecture hybride à deux niveaux développée par la NASA pour des robots d'exploration spatiale. Le premier niveau, le niveau fonctionnel, permet de créer une interface entre l'architecture logicielle et l'architecture matérielle. Le deuxième niveau quant à lui contient le contrôleur d'exécution et la structure décisionnelle comme le montre la figure 1.3. Au sein de chaque niveau, les

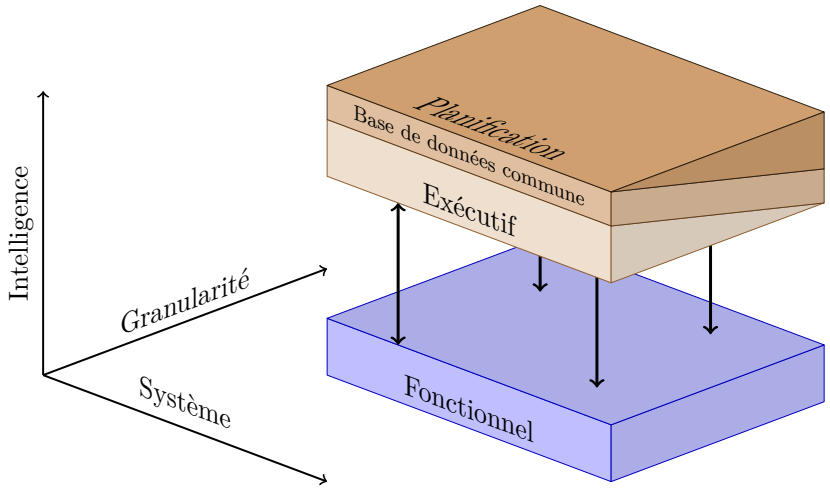


Figure 1.3 – Structure de l'architecture *CLARAty*

différents éléments sont structurés selon une organisation hiérarchique. Cette organisation permet d'abstraire les capacités bas niveau de l'architecture de contrôle des fonctionnalités de haut niveau. Cependant, cela n'empêche pas aux fonctionnalités de haut niveau d'accéder directement aux ressources élémentaires si un contrôle spécifique est nécessaire.

La couche fonctionnelle comporte l'ensemble des fonctions nécessaires au fonctionnement des actionneurs et des capteurs du robot. Différents niveaux de granularité de ces fonctions permettent au robot d'effectuer un ensemble de tâches complexes par l'intermédiaire de planificateurs locaux. En outre, une entité de la couche fonctionnelle se charge

de stocker et de maintenir l'état du système, y compris l'état futur par l'intermédiaire d'un prédicteur d'utilisation de ressources.

La couche décisionnelle quant à elle contient l'« intelligence » du robot. Cette intelligence est donnée par la présence de planificateurs permettant la découpe d'objectifs de haut niveau en « sous-objectifs » classés hiérarchiquement et contraints temporellement. Cet ensemble d'objectifs compose un arbre de tâches organisées de façon séquentielle.

1.1.4 Conclusion sur les architectures logicielles

Il existe un ensemble d'architectures logicielles différentes et possédant des structures internes diverses. Le type d'architecture utilisé au sein d'un robot dépend de plusieurs paramètres comme la mission à accomplir, l'adaptabilité et la réactivité du robot face à des événements imprévus ou les ressources disponibles sur son processeur.

Dans tous les cas, les architectures robotiques sont implémentées sur des calculateurs par l'intermédiaire de tâches. Cependant la complexité des architectures robotiques nécessite bien souvent l'utilisation de *middlewares* afin d'en simplifier l'implémentation.

1.2 Middlewares

Les architectures robotiques ont besoin de s'exécuter sur un calculateur ; pour cela, elles peuvent être implantées sur le système d'exploitation sous la forme de tâches. Cependant, écrire directement des tâches adaptées à un système d'exploitation est un processus fastidieux. Afin de simplifier le développement d'architectures robotiques, des *middlewares* sont souvent utilisés. Les middlewares sont des entités logicielles servant d'interface entre une application et le système d'exploitation afin d'en abstraire les spécificités.

Il existe des middlewares adaptés à différentes situations : les middlewares dédiés à la robotique en simplifiant l'implantation d'architecture complexes ; les middlewares *temps-réel* destinés aux applications dont le bon fonctionnement est fortement liée au temps ; ou ceux possédant un protocole de communication standardisé fournissant des garanties sur la communication de données entre les éléments de l'architecture.

1.2.1 Middlewares robotiques

Les middlewares robotiques ont été avant tout conçus pour des applications robotiques. Ils permettent de simplifier la conception d'architectures robotiques à base de composants

en fournissant des interfaces de communication et la possibilité de définir les activités des tâches. Parmi les middlewares robotiques nous pouvons citer *Robot Operating System (ROS)* [Quigley et al. 2009] ou *Generator of Modules ($G^{en}{}_oM$)* [Mallet, Pasteur et Herrb 2010]. En pratique, le middleware robotique le plus utilisé est *ROS*. Nous détaillons donc ce middleware.

1.2.1.1 *Robot Operating System* (ROS)

ROS [Quigley et al. 2009] est un middleware à base de tâches conçu pour des applications robotiques. Son développement a été orienté pour répondre à trois objectifs : posséder un support multi-langages, comporter un ensemble d'outils pour simplifier le développement et être « léger ».

Lors du développement de codes logiciels, chaque programmeur possède sa propre préférence de langage de programmation. Afin de pouvoir communiquer entre les différents langages de programmation, ROS implémente une couche de communication pair à pair générique utilisant des données sérialisées. La sérialisation des données est effectuée au travers d'une interface de sérialisation/désérialisation implémentée dans les différents langages supportés.

Le middleware ROS s'articule autour de quatre concepts fondamentaux : les *nœuds*, les *messages*, les *topics* et les *services*. Les *nœuds* sont les processus fonctionnels constituants l'architecture logicielle.

Les nœuds communiquent entre eux par l'intermédiaire de *messages* constitués de structures de données strictement typées. Les types de données standards sont supportés comme des entiers, des booléens ou des nombres à virgule flottante, ainsi que des types complexes comme tableaux de données ou des compositions de messages.

La communication entre nœuds s'effectue en connectant des *topics* grâce à un mécanisme de publication/souscription asynchrone. Lorsqu'un nœud émet une donnée, il la publie sur un topic ; pour recevoir une donnée, il doit d'abord s'abonner à un topic portant cette donnée. Un topic peut posséder plusieurs publicateurs et plusieurs souscripteurs et un nœud peut publier ou s'abonner à plusieurs topics. En général les publicateurs et les souscripteurs ne connaissent pas l'existence des autres nœuds.

Lorsque la communication entre nœud nécessite d'être synchronisée, le mécanisme utilisé consiste à la mise en place de *services*. Un nœud peut faire appel à un service fournit par un autre nœud et attend la réponse avant de continuer son exécution.

ROS apporte un ensemble d'outils afin de simplifier le développement d'architectures robotiques. Ces outils permettent par exemple de naviguer dans les sources des codes, de

configurer ou d'obtenir des paramètres de configuration, de visualiser les connections entre pairs, de mesurer l'utilisation de bande passante pour le transfert de données, d'afficher graphiquement des évolutions de données ou encore de générer automatiquement de la documentation.

La « légereté » du middleware définit sa capacité à réutiliser des fonctions et des algorithmes déjà existants sans avoir à les réimplémenter. La philosophie de ROS est d'utiliser le plus possible des librairies externes, indépendantes de ROS, et en les encapsulant dans des tâches.

ROS est utilisé dans de nombreuses applications robotiques comme un système de transport autonome de fruits dans des vergers [Freitas et al. 2012] ou *The Office Marathon* [Beeden et Bruin 2010].

1.2.2 Middlewares temps-réel

Un middleware temps-réel a pour rôle de fournir une interface au développeur afin d'abstraire les spécificités des systèmes d'exploitation d'un point de vue de la gestion temporelle des tâches. Cette abstraction permet au développeur de ne se concentrer que sur les activités des tâches et leur rôle. Parmi les middlewares temps-réel utilisés en robotique, nous pouvons citer *cisst* [Jung et al. 2013], *RTjava* [Bollella 2000], *openRTM* [Ando et al. 2011] ou encore *Orocos* [Soetens et Bruyninckx 2005]. A titre d'exemple, nous détaillons uniquement les middlewares *BIP* et *Orocos*.

1.2.2.1 Orocos

Le middleware *Orocos* [Soetens et Bruyninckx 2005] est un middleware temps-réel destiné à servir de plate forme commune pour des boucles de contrôle temps-réel. Il fournit une boîte à outils temps-réel, le *Real-Time Toolkit (RTT)*, pour gérer l'exécution et l'interaction de composants. Les composants sont des entités logicielles, rattachés à des tâches du système d'exploitation, offrant des services à l'architecture. Ils peuvent aussi bien servir d'interface avec des parties matérielles du système qu'intégrer des algorithmes de calcul. La connexion et l'activation de composants permet de concevoir une architecture.

L'interface des composants est définie par trois éléments, comme le montre la figure 1.4 : les ports de données, les propriétés et les opérations.

— les *ports de données* : ils permettent aux composants de communiquer des données. Il existe deux types de ports : les ports d'entrée et les ports de sortie. Dans les deux cas, ils peuvent contenir un tampon ou pas.

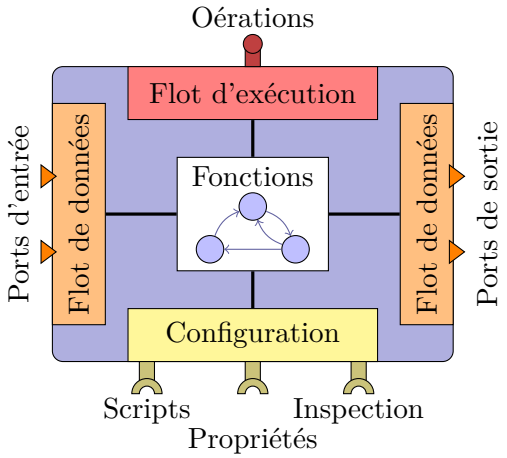


FIGURE 1.4 – Structure d'un composant Orocos

— les *propriétés* : elles permettent de paramétrer les composants. Les propriétés peuvent être chargées lors de la configuration du composant ou modifiées pendant son exécution.
— les *opérations* : ce sont des fonctions propres à un composant, mais appelées par d'autres de manière synchrone ou asynchrone, c'est à dire par le composant appelant la fonction ou par le composant possédant cette fonction. Les opérations sont la plupart du temps utilisées pour obtenir des valeurs de paramètre ou effectuer des calculs "instantanés".

Le middleware Orocos est utilisé dans de nombreux projets, dont le véhicule autonome *Spirit of Berlin* [BERLIN 2007], un générateur de mouvements basé contraintes utilisant deux bras manipulateurs *iTASC* [SMITS et al. 2009] ou encore un système de détection et de reconnaissance multi-cible par drones aériens [CHANEL P. CARVALHO, TEICHTEIL-KÖNIGSBUCH et LESIRE 2013].

1.2.3 Conclusion sur les middlewares

Quelque soit la spécialisation des middlewares, ils permettent au développeur de simplifier la conception d'architectures logicielles en permettant l'abstraction des spécificités du système d'exploitation et du calculateur sur lequel il s'exécute. Cette simplification est principalement apportée par l'homogénéisation de l'implémentation des fonctionnalités de l'architecture logicielle. L'implémentation s'effectue au travers d'une *interface de programmation* spécifique au middleware qui définit une modélisation implicite des architectures. Cependant, quelque soit la spécialisation du middleware, il ne peut pas garantir le fonctionnement correct de l'architecture. Les middlewares temps-réel, par exemple, permettent d'exécuter un ensemble de tâches possédant des contraintes temps-réel mais

ne garantissent pas la capacité d'ordonnancement de telles architectures.

Pour permettre l'analyse de systèmes logiciels, il faut raffiner la modélisation implicite fournie par les middlewares. Pour cela, il existe des langages de modélisation dédiés conçus pour abstraire les aspects bas niveau de l'implémentation et se focaliser sur la modélisation haut niveau. La section 1.3 présente donc un ensemble de langages de modélisation.

1.3 Langage de modélisation

La conception d'architectures logicielles est un processus complexe nécessitant un développement rigoureux. Afin de simplifier la conception de telles architectures, nous avons recours à des langages de modélisation pour abstraire les spécificités de l'implémentation des architectures et permettre d'analyser leur comportement. Il existe différents langages de modélisation apportant des abstractions adaptées à différentes analyses.

1.3.1 Langage de modélisation orienté informatique

Les langages de modélisation orientés informatique se focalisent sur l'abstraction de l'implémentation des systèmes logiciels pour se concentrer sur leur aspect fonctionnel. Ces langages intègrent souvent des outils de génération de code et d'analyse des modèles afin de faciliter la conception d'architectures complexes.

1.3.1.1 *Unified Modeling Language*

Unified Modeling Language, ou *UML*, est un langage de modélisation graphique possédant une syntaxe et une sémantique semi-formelle. Il est structuré par une architecture à quatre couches : la couche *méta-méta modèle* déclare un langage pour spécifier le *méta modèle*, la couche *méta modèle* définit la spécification dans un langage de modélisation spécifique, la couche *modèle* est utilisée pour définir des modèles de systèmes logiciels et enfin la couche *objets utilisateur* permet de construire des instances de modèles.

Les couches *modèle* et *méta modèle* sont les plus importantes pour modéliser des architectures logicielles en UML. Les diagrammes UML peuvent être séparés en deux catégories : la première représente la structure de l'architecture et la seconde décrit les comportements.

La structure de l'architecture se compose de six diagrammes distincts : les *diagrammes de classe* représentent les classes système, les attributs et les relations entre classes,

les *diagrammes de composants* décrivent les interdépendances entre composants, les *diagrammes de déploiements* représentent le matériel utilisé lors de l'implémentation, les *diagrammes de structure* décrivent la structure interne des classes, les *diagrammes d'objets* représentent la structure complète ou partielle du système modélisé et enfin les *diagrammes de paquets* décrivent le découpage d'un système en groupes et les dépendances entre groupes.

Les fonctionnalités du système logiciel sont dictées par un ensemble de sept diagrammes comportementaux : les *diagrammes d'activité* représentent les étapes d'exécution des composants opérationnels, les *diagrammes de cas d'étude* décrivent les fonctionnalités d'un système en termes d'acteurs, d'objectifs et de dépendances parmi les cas d'étude, les *diagrammes de machine à états* représentent les états et les transitions entre états, les *diagrammes de communication* représentent les interactions entre objets sous la forme de messages séquentiels, les *diagrammes temporels* représentent les contraintes temporelles, les *diagrammes d'interaction* fournissent une vue d'ensemble des nœuds représentant les diagrammes de communication et enfin les *diagrammes de séquence* représentent les communications entre objets.

De plus, le langage UML peut être adapté grâce à l'utilisation de *profils*. La spécification d'un profil ajoute des capacités spécifiques aux outils fournis par UML. Les profils permettent donc d'étendre le langage de modélisation ou d'intégrer de nouvelles méthodes d'analyse. Le profil *MARTE* [Selic et Gérard 2013] permet par exemple de fournir des fonctionnalités de développement de systèmes embarqués temps-réel en remplaçant les profils UML existants *ordonnançabilité*, *performance* et *temps*. Le profil MARTE est spécialisé à la fois pour la modélisation et l'analyse de systèmes temps-réel. La partie de modélisation permet de définir la spécification initiale d'un système temps-réel jusqu'aux caractéristiques détaillées de l'architecture embarquée du système. La partie d'analyse se focalise sur des analyses basées modèles afin de supporter des techniques d'analyse existantes. Pour cela, MARTE permet d'annoter les modèles afin d'y ajouter les informations nécessaires aux analyses, en particulier pour des analyses de performance et d'ordonnançabilité.

1.3.1.2 *Architecture Analysis & Design Language*

Le langage *Architecture Analysis & Design Language (AADL)* [Feiler, Gluch et Hudak 2006] est un langage de modélisation conçu pour la spécification, l'analyse, l'intégration automatique et la génération de code pour des systèmes de calculs distribués temps-réel. Par l'intermédiaire d'extensions, AADL fournit un moyen *suffisamment précis* de modéliser les architectures de systèmes temps réel embarqués, tels que des systèmes avioniques [Feiler, Gluch, Hudak et Lewis 2004] ou des unités de contrôle automobiles [Shiraishi 2010]. Cela permet l'analyse des propriétés du système modélisé

et une implémentation prédictible lors de l'intégration du système par l'intermédiaire d'outils de génération de code.

AADL est un langage à la fois textuel et graphique utilisé pour modéliser et analyser les architectures logicielles et matérielles de systèmes embarqués. Le fonctionnement de ces systèmes dépend fortement de besoins non fonctionnels tels que la fiabilité, la disponibilité, la sûreté ou le comportement temporel. AADL est utilisé pour décrire la structure de systèmes comme un assemblage de composants logiciels attachés à une plate-forme d'exécution. Il peut être utilisé pour décrire les interfaces fonctionnelles des composants, comme les ports d'entrée et de sortie de données, et des aspects de performances, comme le comportement temporel. AADL peut aussi être utilisé pour décrire les interactions entre composants, comme par exemple les connexions entre composants ou l'allocation des composants sur la plate-forme d'exécution. Le langage peut aussi être utilisé pour décrire le comportement dynamique de l'exécution de l'architecture en permettant de modéliser les modes opérationnels et les transitions de modes du système logiciel.

De plus, AADL est un langage de modélisation extensible conçu pour accepter des analyses que le langage de base ne supporte pas complètement. Ces extensions peuvent prendre la forme de nouvelles propriétés et de notations spécifiques aux analyses qui sont associées aux composants, comme l'annexe comportementale qui est dédiée à étendre les capacités d'analyse temps-réel existantes.

1.3.1.3 *Cyber-Physical Action Language*

Le langage *Cyber-Physical Action Language* [Altmeyer, Navet et Fejoz 2015], ou *CPAL*, est un langage de modélisation interprété conçu pour fournir des abstractions haut niveau pour des systèmes embarqués et fournir des garanties sur le comportement temporel de l'application. Contrairement aux autres langages de modélisation, CPAL n'intègre pas de fonctionnalités de génération de code : les modèles sont directement exécutés au travers d'un interpréteur.

Les abstractions haut niveau apportées par CPAL se situent sur trois éléments principaux : les mécanismes d'ordonnancement temps réel, le comportement des processus et les communications entre processus.

— les mécanismes d'ordonnancement temps réel : les processus d'un système peuvent être activés soit par un mécanisme d'exécution périodique, soit par l'arrivée d'un événement externe ;
— le comportement des processus : le comportement logique des processus est défini par des machines à états finis au sein desquelles le code peut être exécuté soit dans les états, soit lors des transitions ;
— les communications entre processus : les différents processus d'un système commu-

niquent entre eux et avec les interfaces matérielles par l'intermédiaire de ports de communication possédant des sémantiques bien définies, comme la mise des données en mémoire tampon par exemple.

L'interpréteur de modèles de CPAL supporte deux types d'exécution : la première consiste à installer l'interpréteur sur un système d'exploitation, ce qui permet à l'application de bénéficier des interfaces matérielles et logicielles fournies par celui-ci. La contrepartie de ce mode d'interprétation est l'incertitude d'exécution liée au fonctionnement du système d'exploitation et des préemptions qu'il peut apporter à l'application modélisée. Le second mode d'interprétation permet de pallier à ce problème en se passant du système d'exploitation : l'interpréteur est directement implémenté sur la cible matérielle, garantissant le comportement temporel de l'exécution du modèle.

1.3.2 Langage de modélisation orienté robotique

Les langages de modélisation orientés robotique reprennent les caractéristiques principales des langages orientés informatique, comme les niveaux d'abstraction et les outils de génération de code et d'analyse. Ils sont cependant moins généralistes et se focalisent sur la modélisation du système logiciel de robots ainsi que des interactions entre le logiciel et le matériel.

1.3.2.1 *Robotic Modeling Language*

Robotic Modeling Language (RobotML) [Dhouib et al. 2012] est un langage de modélisation destiné à la robotique. Il est constitué de quatre éléments principaux : l'architecture, le comportement, la communication et le déploiement. De plus RobotML fournit au développeur des outils de génération de code exécutable.

L'architecture est, elle même, constituée de cinq éléments distincts : le *système robotique* définissant les différents composants logiciels, le *système environnement* décrivant l'environnement dans lequel évolue les robots, les *types de données* listant les différents types d'informations échangées au sein de l'architecture, la *mission robotique* décrivant l'ensemble des objectifs devant être atteints par le robot et la *plate-forme* représentant les divers exécutifs robotiques tels que les middlewares, les simulateurs ou la plate-forme matérielle du robot.

Le comportement des composants est décrit, soit par des algorithmes individuels, soit par des machines à états. La description du comportement des composants sous le forme de machine à états permet une représentation plus précise du fonctionnement au sein du modèle.

Les communications permettent de formaliser les échanges de données et les requêtes de services entre systèmes robotiques. Elles peuvent se matérialiser sous la forme de *ports* correspondant au modèle publication/souscription ou sous la forme de *service* mettant en place un principe de requête/réponse.

Le déploiement contient un ensemble d'outils permettant d'assigner un système robotique à une plate-forme, comme un middleware ou un simulateur. Le déploiement est aussi initiateur des générateurs de code à partir des paramètres spécifiques des plates-formes.

1.3.2.2 *Behavior, Interaction, Priority* (BIP)

Behavior, Interaction, Priority [Basu, Bozga et Sifakis 2006], plus communément appelé *BIP* est un cadre de modélisation pour des tâches temps-réel hétérogènes. Le modèle de composant associé est constitué d'une superposition de trois couches : la couche basse décrit le comportement fonctionnel du composant comme un ensemble de transitions formant une machine à états finis ; la couche intermédiaire contient les connecteurs permettant les interactions entre les transitions de la couche basse et la couche haute ; la couche haute consiste en un ensemble de règles de priorité pour décrire les politiques d'ordonnancement. Ce découpage permet une séparation claire entre le comportement fonctionnel des composants et la structure de leurs interactions.

Le langage BIP est constitué d'un langage de modélisation et d'un ensemble d'outils permettant d'éditer les modèles BIP ainsi que de les analyser. La plate-forme d'analyse consiste en un moteur de simulation et une infrastructure logicielle pour exécuter des traces de simulation à partir des modèles de composants. De plus, la plate-forme d'analyse permet d'explorer l'espace d'état des machines à états pour s'assurer de la validité de propriétés comme l'absence de blocages mortels ou la présence d'invariants d'états.

Enfin BIP contient un moteur d'exécution permettant de générer du code C++ et de l'exécuter selon les spécifications données dans les modèles de composants.

Le langage BIP est notamment utilisé dans des applications de robotique autonome en conjonction avec $GenoM$ [Abdellatif et al. 2012 ; Basu, Gallien et al. 2008]

1.3.2.3 *3-View Component Meta-Model*

3-View Component Meta-Model [Alonso et al. 2010], ou *V3CMM*, est un langage de modélisation indépendant de la plate-forme pour la conception d'architectures logicielles à base de composants. Les principaux concepts de V3CMM sont articulés autour de trois *vues* principales : le méta-modèle, la vue structurelle et la vue comportementale.

La structure du méta-modèle se focalise sur la simplicité et la réutilisation de composants. La simplicité est obtenue en n'utilisant qu'un nombre limité de concepts : V3CMM utilise un ensemble minimal de concepts permettant la modélisation d'applications à base de composants réutilisables. Pour cela, V3CMM comprend trois concepts complémentaires : la *structure* qui décrit l'organisation statique de composants simples ou complexes, la *coordination* définissant le comportement événementiel des composants et l'*algorithme* représentant les fonctions exécutées par les composants. La réutilisation des composants est rendue possible grâce à deux facettes : la réutilisation d'éléments comportementaux et structuraux ainsi que la conception d'architectures indépendantes de la plate forme.

La vue structurelle de V3CMM a pour but d'organiser les différents composants de l'architecture. Pour cela deux concepts sont définis : la définition de composants et l'instanciation de composants. La définition d'un composant permet de déclarer les ports et les interfaces de données représentant respectivement les points de connexion et les définitions des messages échangés. L'instanciation des composants représente l'implantation réelle d'un composant au sein d'une architecture.

Enfin la vue comportementale est articulée autour des machines à états UML. Cependant, comme pour la vue structurelle, V3CMM distingue deux concepts pour les machines à états : leur définition et leur instanciation. La définition d'une machine à états consiste à déclarer ses différents états et transitions. L'instanciation d'une machine à états est quant à elle effectuée lors de l'instanciation des composants.

1.3.3 Conclusion sur les langages de modélisation

Les langages de modélisation, qu'ils soient destinés à des applications robotiques ou informatiques, ont un objectif commun : simplifier la conception de systèmes logiciels complexes. Ils permettent d'abstraire les étapes d'implémentation des fonctionnalités logicielles et matérielles en simulant le fonctionnement et les interactions entre les différents modules composant le système modélisé. De plus, les langages de modélisation peuvent être liés à des générateurs de code afin de transcrire les modèles en codes exécutables tout en réduisant les erreurs d'implémentation. Enfin, les langages de modélisation sont souvent associés à des outils d'analyse permettant de valider les différents aspects fonctionnels et temporels des systèmes.

Les outils de validation et plus particulièrement d'analyse temps-réel étant des utilitaires complexes, ils sont bien souvent déportés des langages de modélisation dans des outils spécialisés. La section suivante présente les principes régissant le fonctionnement et les différents types d'analyse temps-réel.

1.4 Systèmes temps-réel

Un système temps-réel consiste en un système logiciel interagissant avec un procédé externe dont l'exécution est régie par le temps. Le système temps-réel possède donc des contraintes temporelles plus ou moins strictes selon la criticité du procédé avec lequel il interagit.

1.4.1 Classification des systèmes temps-réel

Les contraintes liées aux systèmes temps-réel varient en fonction des besoins des applications auxquelles ils sont confrontés. Les systèmes temps-réel sont classés selon trois catégories principales selon les contraintes temporelles de ces systèmes :

- **Temps-réel dur :** Les contraintes temporelles de tels systèmes doivent obligatoirement être respectées. Le non-respect des contraintes temporelles peut provoquer des comportements imprévisibles et/ou catastrophiques.
- **Temps-réel souple :** Les systèmes temps-réel souples tolèrent le dépassement des contraintes temporelles. Lorsque ces contraintes ne sont pas respectées, la qualité de service du système est dégradée sans pour autant nuire à l'exécution immédiate.
- **Temps-réel hybrides :** Les systèmes temps-réel hybrides sont des systèmes mêlant à la fois des contraintes temporelles dures et souples.

1.4.2 Mécanismes d'ordonnancement

Un système logiciel complexe est souvent constitué d'un ensemble de tâches contenant des fonctionnalités indépendantes s'exécutant en parallèle au sein d'une architecture logicielle. Le bon fonctionnement d'un tel système nécessite un mécanisme afin d'ordonnancer les différentes tâches. Pour cela, l'ordonnanceur organise l'exécution des tâches selon différents paramètres, comme la période ou la priorité des tâches, afin de permettre à chaque tâche de s'exécuter selon ses spécifications.

Pour un ensemble de tâches donné, s'il existe une solution d'ordonnancement et qu'un ordonnanceur est capable de la trouver, alors cet ordonnanceur est optimal. De plus si un ordonnanceur optimal ne peut pas trouver d'ordonnancement valide pour un ensemble de tâches, alors aucun autre ordonnanceur ne peut trouver de solution.

1.4.2.1 Modèle canonique de tâche

Une tâche canonique est définie comme un élément logiciel insécable, préemptible et exécuté périodiquement. Le modèle de tâche associé à ces tâches canoniques est constitué de trois éléments : un temps d'exécution C, une période T et une échéance relative D. Le temps d'exécution correspond au temps passé par le processeur à calculer les fonctions contenues au sein d'une tâche. La période représente le délai entre deux exécutions successives d'une tâche. L'échéance relative définit la durée maximale d'exécution entre la date de début d'exécution d'une tâche et sa date de fin. Une tâche τ peut donc être représenté par le triplet (C, T, D). Il arrive parfois que l'échéance soit définie égale à la période et ne fasse pas partie du modèle de tâche.

La définition d'une tâche est unique mais son exécution est plurale. Lorsqu'une tâche est exécutée, elle est alors appelée *instance de tâche*. Chaque instance de tâche possède les mêmes paramètres que la définition du modèle canonique ; la seule différence concerne la date de réveil R de chaque instance. La figure 1.5 représente l'exécution temporelle de deux *instances* d'une même tâche τ_i.

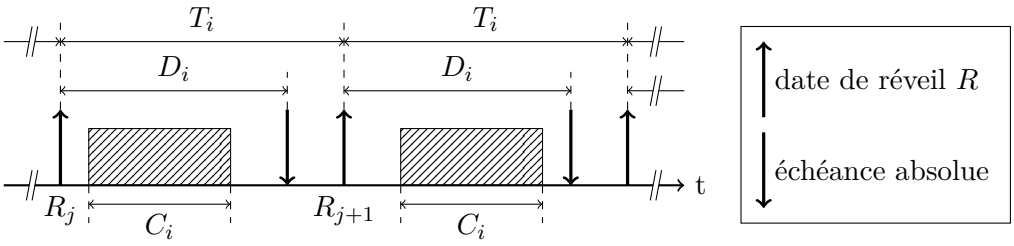


FIGURE 1.5 – Modèle canonique de tâche périodique

Selon les algorithmes d'ordonnancement ou les méthodes d'analyse utilisés sur le système de tâches, les échéances relatives sont complétées par les échéances absolues : l'échéance absolue d'une tâche représente la date à laquelle elle doit impérativement terminer son exécution.

D'autres modèles de tâche existent mais ne sont pas concernés par ces travaux de thèse : les tâches pseudo-périodiques et les tâches apériodiques. Les tâches pseudo-périodiques sont une généralisation des tâches périodiques : elles ne possèdent pas de période spécifique mais un intervalle de temps minimal entre deux exécutions d'une tâche. Les tâches apériodiques sont des tâches activées par l'arrivée d'événements extérieurs au système de tâche. Leurs instants d'activation sont donc inconnus mais leur traitement doit être effectué avant une échéance temporelle. Ces tâches sont bien souvent associées à des comportements nécessitant un traitement urgent.

1.4.2.2 Algorithmes d'ordonnancement

Il existe deux catégories de stratégies d'ordonnancement et d'analyse de systèmes de tâches simples : les ordonnanceurs à priorités fixes et les ordonnanceurs à priorités dynamiques.

Ordonnanceurs à priorités fixes Un ordonnanceur est dit à priorités fixes si toutes les instances d'une même tâche s'exécutent avec la même priorité. La priorité des tâches est définie hors ligne selon des critères de criticité ou des critères temporels. Les deux principaux algorithmes d'ordonnancement à priorité fixe sont *Deadline Monotonic* et *Rate Monotonic*.

Deadline Monotonic L'algorithme *Deadline Monotonic*, proposé dans [Leung et Whitehead 1982] permet de traiter les tâches dont les échéances sont inférieures ou égales à leur période. Ces travaux étendent ceux de [Liu, James et Layland 1973] en levant la contrainte d'échéance sur requête.

La priorité d'une tâche est inversement proportionnelle à son échéance relative. Ainsi la tâche dont l'échéance relative est la plus courte possède la priorité la plus élevée. Deux tâches ayant la même échéance relative possèdent la même priorité. Dans cette configuration, si les deux tâches sont éligibles au processeur, la sélection est soit arbitraire, soit selon un sous-algorithme d'ordonnancement.

Le résultat remarquable de ces travaux est l'optimalité du point de vue de l'ordonnançabilité pour des ensembles de tâches périodiques, indépendantes, dans un contexte préemptif et possédant des priorités fixes.

Rate Monotonic L'algorithme *Rate Monotonic* (RM) est un ordonnanceur en ligne à priorité fixe proposé par [Liu, James et Layland 1973]. La priorité d'une tâche est inversement proportionnelle à sa période. Ainsi, la tâche dont la période est la plus faible possède la priorité la plus élevée. Deux tâches ayant la même période obtiennent la même priorité. Dans ce cas, si les deux tâches sont éligibles au processeur, soit la sélection est arbitraire, soit elle suit un sous-algorithme d'ordonnancement [Martin et Minet 2006].

Pour un ensemble de tâches périodiques, indépendantes et à échéances sur requêtes, dans un contexte préemptif, cet algorithme est optimal. Du point de vue du concepteur, sous ces conditions, cet algorithme garantit que les tâches dont la fréquence d'activation est la plus élevée ne seront jamais retardées ni préemptées par des tâches de période plus longue.

Ordonnanceurs à priorités dynamiques Un ordonnanceur est dit à priorités dynamiques si la priorité de chaque instance de tâche est définie en ligne selon des critères de criticité ou des critères temporels. Les deux principaux algorithmes d'ordonnancement à priorité dynamiques sont *Earliest Deadline First* et *Least Laxity First*.

Earliest Deadline First L'algorithme d'ordonnancement *Earliest Deadline First* (EDF) est issu des travaux de [Liu, James et Layland 1973]. La priorité de chaque instance d'une tâche est inversement proportionnelle à son échéance absolue. Ainsi, lors de l'exécution du système, l'ensemble des tâches éligibles au processeur est trié suivant la valeur de l'échéance absolue, la tâche la plus prioritaire étant celle dont l'échéance absolue est la plus proche ; la moins prioritaire, celle dont l'échéance absolue est la plus tardive. Le réveil de toute instance de tâche impose de recalculer l'ensemble des priorités de l'ensemble de tâches éligibles. L'attribution dynamique de priorités est donc fonction du contexte d'exécution.

Pour les systèmes monoprocesseur, dans un contexte préemptif, cet algorithme est optimal pour des tâches indépendantes, périodiques et à échéances sur requêtes. L'optimalité de l'algorithme, a, par la suite, été démontrée pour des tâches non périodiques. L'avantage de cet algorithme, pour le concepteur, est de garantir que la tâche dont l'échéance est la plus proche sera toujours exécutée, ce qui est un avantage certain dans le traitement des tâches apériodiques correspondant, par exemple, à des alarmes pour le système.

Least Laxity First L'algorithme *Least Laxity First* (LLF) [Mok 1983] est un autre membre des algorithmes à priorité dynamique. Le critère de laxité, différence entre l'échéance et le temps restant de calcul à l'instant t, permet d'attribuer la priorité d'exécution de la tâche. Ainsi, la tâche dont la laxité est la plus faible est la plus prioritaire, et la tâche dont la laxité est la plus importante, celle de plus basse priorité.

Le principal inconvénient de LLF réside dans son importante consommation en ressources de calcul. En effet, les priorités doivent être recalculées régulièrement pour prendre en compte l'évolution de la laxité des tâches éligibles. De plus, cet algorithme tend à augmenter le nombre de changements de contexte, source supplémentaire de consommation de ressources. Ainsi, si deux tâches ont la même échéance, LLF activera alternativement une tâche puis l'autre à chaque itération de calcul des priorités.

1.4.3 Analyses des systèmes temps-réel

Parmi les différents types d'analyses existantes, nous nous concentrons uniquement sur les analyses temporelles. Les analyses temporelles, aussi appelées analyses temps-réel, servent à vérifier si l'exécution des éléments de l'architecture s'effectue dans le temps imparti. Pour cela, la notion de faisabilité d'un ensemble de tâches est définie : un ensemble de tâches est dit *faisable* si il existe une méthode d'ordonnancement capable d'ordonnancer cet ensemble de tâches.

Afin de garantir qu'un ensemble de tâches est faisable, plusieurs travaux ont démontré l'existence de bornes supérieures d'utilisation processeur garantissant la faisabilité d'un ensemble de tâches. L'utilisation processeur U est définie comme la proportion de temps passé alloué à un ensemble de tâches sur le processeur. Dans le cas d'un ensemble de n tâches ordonnancées par un algorithme d'ordonnancement à priorités fixes comme RM, la borne d'utilisation maximale est calculée par l'équation 1.1 présentée par dans les travaux de [Liu, James et Layland 1973].

$$U = \sum_{i=1}^{n} \frac{\mathcal{C}_i}{T_i} \tag{1.1}$$

Pour un nombre élevé de tâches, cette borne converge vers une valeur de taux d'utilisation de 69 %. Ces travaux ont été étendus aux algorithmes d'ordonnancement à priorité dynamique [Dertouzos 1974] comme EDF pour lequel la borne maximale d'utilisation processeur assurant la faisabilité de l'ensemble de tâches est de 100 %.

La faisabilité d'un ensemble de tâches est une notion utile afin de déterminer l'ordonnançabilité de cet ensemble. Cependant le calcul de la borne maximale d'utilisation processeur est pessimiste et certains ensembles de tâches peuvent être ordonnançables tout en ayant un taux d'utilisation processeur supérieur à la borne maximale.

Afin de déterminer plus précisément les capacités d'ordonnancement d'un ensemble de tâches, des méthodes basées sur le calcul du pire temps de réponse des tâches ont été mises en place. Il existe différents types d'analyse selon le type de tâche impliquées dans les architectures : des analyses pour des tâches canoniques, pour des tâches partageant des ressources, pour des tâches interdépendantes, ou encore pour des tâches représentées sous forme de graphes.

1.4.3.1 Tâches canoniques

Le principe d'analyse d'un système de tâches canoniques se base sur le calcul du pire temps de réponse $\mathcal{R}$ des différentes tâches. Les tâches du système sont dites ordonnançables

si leur pire temps de réponse est inférieur à leur échéance. Le temps de réponse d'une tâche correspond au délai entre le début et la fin d'exécution de la tâche en prenant en compte les préemptions des autres tâches du système. Ce calcul se base sur les priorités calculées par les algorithmes d'ordonnancement.

Le calcul du pire temps de réponse des tâches se base sur un processus récursif rappelé dans l'équation 1.2.

$$\begin{aligned} \mathcal{R}_i^0 &= 0 \\ \mathcal{R}_i^{n+1} &= C_i + \sum_{j \leq hp(i)} \left\lceil \frac{\mathcal{R}_i^n}{T_j} \right\rceil \mathcal{C}_j \end{aligned} \tag{1.2}$$

où C_i représente le temps d'exécution de la tâche i et $hp(i)$ donne les instances de tâches de plus haute priorité que la tâche i.

Deux conditions peuvent arrêter cette boucle récursive :
— lorsque deux itérations consécutives ont la même valeur $\mathcal{R}_i^n = \mathcal{R}_i^{n+1}$;
— lorsque $\mathcal{R}_i^{n+1}$ atteint l'échéance de la tâche i.
Dans le second cas, la tâche n'est pas ordonnançable. Si le pire temps de réponse de chaque tâche de l'architecture est inférieur à son échéance ($\forall i, \mathcal{R}_i \leq D_i$), le système est ordonnançable.

1.4.3.2 Autres modèles de tâche

Le modèle canonique de tâches permet de modéliser des ensembles de tâches simples mais la complexité des applications augmentant, ce modèle de tâche ne suffit plus. Par conséquent, nous avons besoin de modèles de tâches prenant en compte plus précisément le fonctionnement réel du système, comme par exemple la gestion de ressources partagées, la dépendance entre tâches ou la structure interne des tâches.

Modèle de tâche partageant des ressources Le problème de partage de ressources se pose lorsque plusieurs tâches doivent accéder à une ou plusieurs ressources protégées par des sémaphores ou par des mécanismes d'exclusion mutuelle. Les ressources partagées peuvent correspondre à une ressource matérielle ou à de la mémoire partagée ; par exemple, l'accès concurrent à ce type de ressource peut entraîner des comportements non-déterministes. Cependant, les mécanismes de protection des ressources partagées peuvent dans certaines conditions provoquer le blocage des tâches : si deux tâches τ_1 et τ_2 doivent accéder à la même ressource et que τ_1 est plus prioritaire que τ_2, il est possible que τ_2 commence son exécution et réserve la ressource. Ensuite τ_1 débute à son tour son exécution et préempte τ_2 avant qu'elle n'ait pu libérer la ressource. Lorsque τ_1 aura à son tour besoin de la ressource, elle ne pourra pas y accéder et l'exécution des deux tâches

sera interrompue.

Pour palier à ce phénomène d'inter-blocage des tâches, différents mécanismes d'ordonnancement ont été mis en place, comme *Priority Inhreitance Protocol*, *Priority Ceiling Protocol* ou *Stack Ressource Policy*.

Priority Inhreitance Protocol Le protocole *Priority Inhreitance Protocol* [Sha, Rajkumar et Lehoczky 1990] ou PIP est une technique d'ordonnancement pour des tâches utilisant des ressources partagées. Lorsque plusieurs tâches partagent de manière exclusive une ressource, elles utilisent un mécanisme de sémaphore afin de réserver, puis de libérer, la ressource. Un problème survient lorsqu'une tâche de faible priorité réserve une ressource puis qu'une tâche de priorité plus élevée demande cette même ressource. La tâche de haute priorité doit attendre que la tâche de priorité basse rende la ressource avant de pouvoir poursuivre son exécution sachant que la tâche de priorité basse peut, elle même, être préemptée par d'autres tâches. Ce mécanisme s'appelle l'*inversion de priorité*.

Afin de palier à ce problème, au sein d'un ensemble de tâches partageant une ou plusieurs ressources, les tâches de priorités basses héritent de la priorité de la tâche la plus prioritaire lors du verrouillage du sémaphore concernant la ressource. Ces tâches reprennent leur priorité nominale lorsqu'elles libèrent le sémaphore. Ce mécanisme d'héritage de priorité permet de borner le temps de préemption d'une tâche partageant une ressource.

Priority Ceiling Protocol Le protocole *Priority Ceiling Protocol* [M.-I. Chen et Lin 1990] ou PCP est une technique d'ordonnancement pour des tâches partageant des ressources. Cette méthode a été développée afin de palier aux limites de PIP. Le protocole PIP adresse le problème d'inversion de priorité mais possède deux limites : le blocage mortel de deux tâches et la chaîne de blocage.

Le protocole PCP fonctionne de la manière suivante : chaque tâche τ peut utiliser une ressource en la protégeant par un sémaphore s. De plus, chaque sémaphore S_i se voit attribuer une priorité P égale à celle de la tâche de plus haute priorité utilisant cette ressource : $P(S_i) = P(\tau_i)$. Soit une tâche τ ayant la priorité la plus élevée parmi les tâches en attente d'exécution et possédant une section critique protégée par le sémaphore S. Soit S^* le sémaphore bloqué avec la priorité la plus élevée parmi les sémaphores déjà bloqués. Avant d'entrer en section critique, la tâche τ doit verrouiller le sémaphore S. L'exécution de τ peut se dérouler selon deux scénarios :

- si la priorité de τ est moins élevée que S^*, elle ne peux pas verrouiller S et son exécution est bloquée ;
- si la priorité de τ est plus élevée que S^*, elle verrouille S et s'exécute, bloquant

les autres tâches.
Ce protocole s'ajoute à celui de PIP afin de permettre l'héritage de priorités des tâches utilisant un sémaphore. PCP permet donc de borner le temps de préemption d'une tâche partageant une ressource lors d'inversions de priorités mais aussi lors de blocages mortels ou de chaînes de blocage.

Stack Ressource Policy Le protocole *Stack Ressource Policy* [Baker 1991] ou SRP est une technique d'ordonnancement permettant de généraliser le protocole PCP.

Le protocole SRP se base sur un fonctionnement similaire à PCP. Cependant, un niveau de préemption est associé à chaque tâche. Les niveaux de préemption sont attribués en fonction des échéances des tâches : plus l'échéance est courte, plus le niveau de préemption est élevé. Lors de l'exécution, une priorité est attribuée à chaque ressource partagée telle que la priorité est égale au niveau de préemption maximal des tâches partageant cette ressource. Lorsqu'une tâche est réveillée, elle ne peut préempter la tâche courante que si son échéance absolue est plus courte et si son niveau de préemption est plus élevé que la plus haute priorité des ressources verrouillées.

Modèle de tâches interdépendantes Le problème des tâches interdépendantes se pose lorsque l'exécution de tâches est déterminée par d'autres tâches. Les tâches interdépendantes se trouvent souvent sous la forme de tâches sporadiques dont le réveil est lié au résultat de l'exécution d'autres tâches, dans des systèmes logiciels comportant à la fois des tâches périodiques et sporadiques. L'ordonnancement de tels ensembles de tâches doit donc être dynamique lors de l'exécution du système sans pour autant gêner le fonctionnement des tâches périodiques.

Dans ce contexte, [Chetto, Silly et Bouchentouf 1990] a proposé une méthode permettant l'ordonnancement dynamique de tâches interdépendantes. Le principe consiste à utiliser des algorithmes d'ordonnancements « classiques » sur des ensembles de tâches dont les paramètres peuvent varier en fonction de l'exécution de tâches interdépendantes. Les deux paramètres des tâches impactés par l'insertion de tâches sporadiques sont l'échéance D et la date de réveil au plus tard R. La stratégie d'ordonnancement se découpe en deux phases : dans un premier temps, de nouvelles échéances et dates de réveil sont calculées et dans un second temps, ces nouveaux paramètres sont utilisés pour déterminer les nouvelles priorités des tâches.

Soit un ensemble de tâches contenant deux tâches interdépendantes τ_i et τ_j telles que l'exécution de τ_i entraîne le réveil de τ_j. Afin que l'ordonnancement soit valide, τ_i doit terminer son exécution avant la date de réveil au plus tard de τ_j. L'échéance D_i de τ_i vaut donc $D_i = \min(D_i, D_j - \mathcal{C}_j)$. La date de réveil au plus tard R_i de τ_i est alors contrainte par l'exécution de τ_j : $R_i \leq \max(R_i, R_j + \mathcal{C}_j)$.

Ensuite, la priorité des tâches est recalculée en prenant en compte les nouvelles échéances et dates de réveil au plus tard des tâches. Enfin, l'ordonnancement est déterminé en utilisant des algorithmes usuels, tels que EDF.

Modèle de tâches à base de graphes Contrairement aux trois autres modèles de tâches, celui permettant de définir des tâches à base de graphes est différent : les tâches ne sont plus représentées sous la forme $(\mathcal{C}, P, T, D)$, mais sous la forme de trois listes temporellement ordonnées d'éléments. Ces listes sont : des temps d'exécutions $\mathcal{C}$, des échéances D et des dates de réveil R. Dans certains cas les dates de réveil peuvent être remplacées par des délais minimaux P entre deux instances d'exécution de la tâche. Cette description permet de représenter l'exécution d'une tâche comme un graphe d'instances de tâche.

Parmi les différents modèles de tâches à base de graphes, nous pouvons citer *Generalized Multiframe* [Baruah, D. Chen et al. 1999], *Digraph Real-Time* [Stigge et al. 2011], *Multiframe* [Mok et D. Chen 1997], *Non-Cyclic Generalized Multiframe* [Tchidjo Moyo et al. 2010] ou *Recurring Real-time* [Baruah 2003]. Ces différents modèles de tâches peuvent hiérarchisés selon leur expressivité et leur complexité d'analyse, comme le montre la figure 1.6.

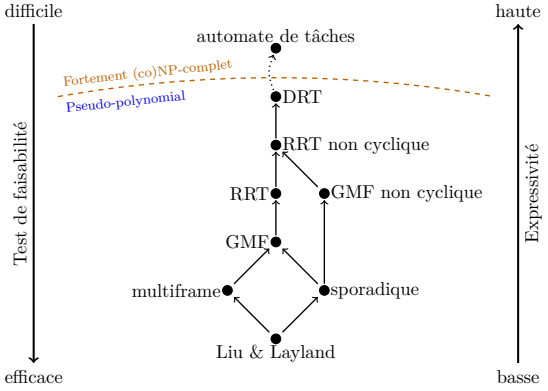


Figure 1.6 – Organisation de différents modèles de tâches

Generalized Multiframe Le modèle de tâche nommé *Generalized Multiframe* [Baruah, D. Chen et al. 1999], ou *GMF*, propose de définir les tâches comme des ensembles ordonnés de travaux, chacun possédant un temps d'exécution, une échéance et un délai minimal entre deux travaux.

Afin de calculer l'ordonnançabilité d'un tel système de tâches, la méthodologie proposée par GMF consiste à calculer une fonction de demande pour chaque tâche. Une

fonction de demande représente le temps d'exécution cumulé pouvant être demandé à un processeur par une tâche ou un ensemble de tâches pendant un intervalle de temps. L'ordonnançabilité du système de tâches est garantie lorsque la somme des fonctions de demande de chaque tâche pour tout instant $rbf(t)$ est inférieure à t.

Digraph Real-Time Le modèle de tâche appelé *Digraph Real-Time* [Stigge et al. 2011], ou *DRT*, généralise l'approche de GMF en caractérisant chaque tâche d'un système logiciel par un *graphe dirigé*. Les nœuds de ce graphe représentent les types de travaux pouvant être générés par la tâche. Chaque nœud est représenté par un pire temps d'exécution et une échéance ; les arcs reliant les nœud définissent le délai minimal entre les exécutions des travaux.

Le principe de cette analyse se base sur le calcul de la *fonction de demande* de l'ensemble des tâches contenues dans le système. La fonction de demande est calculée à partir des différentes exécutions des tâches impliquées dans le système logiciel. La fonction de demande correspond à une borne supérieure des séquences possibles de travaux générés par l'ensemble des tâches. Le processus conclut sur l'ordonnançabilité du système si la fonction de demande à chaque instant $rbf(t)$ est inférieure à t.

1.4.4 Conclusion sur les systèmes temps-réel

Un système temps-réel est un système logiciel complexe soumis à des contraintes temporelles strictes. Les différents types de mécanismes d'ordonnancement permettent de répondre aux niveaux de criticité et aux besoins des applications.

La vérification du bon fonctionnement de telles applications est effectuée *a priori* par des outils d'analyse d'ordonnancement. Ces outils se basent sur des modèles de tâches dont la complexité dépend du rapport entre le niveau de précision de l'analyse et le temps acceptable passé à analyser le système.

1.5 Conclusion

L'objectif de ce chapitre est de présenter les quatre éléments constitutifs d'un système temps-réel pour la robotique : les architectures robotiques, les middlewares, les outils et langages de modélisation, et les analyses temps-réel. Les architectures robotiques modernes embarquent de plus en plus de fonctionnalités au travers de structures à base de composants et les temps de développements deviennent de plus en plus courts. Pour cela, elles basent souvent leur exécution sur des middlewares permettant de simplifier

le développement bas niveau des composants. De plus, des outils de modélisation permettent d'accélérer et de faciliter la conception haut niveau des architectures robotiques. Cependant, pour que ces architectures soient sûres, il faut les analyser et garantir que leur comportement respecte un ensemble de contraintes temporelles et fonctionnelles.

Cet état de l'art nous permet de constater que la conception d'architectures robotiques est un processus complexe sur plusieurs niveaux d'abstraction. Le modèle d'exécution associé à ces architectures est donc suffisamment détaillé pour pouvoir exprimer leur fonctionnement, à la fois d'un point de vue comportemental que temporel. De plus, il existe un ensemble de méthodes d'analyse temps-réel adaptées à des modèles d'exécution assez simples par rapport à la complexité des architectures robotiques ne permettant pas de les analyser correctement. Nous avons donc proposé une méthode d'analyse présentée dans le chapitre suivant.

Chapitre 2

Méthodologie globale

Cette étude s'inscrit dans une démarche d'analyse temporelle du système logiciel de robots autonomes dès la phase de conception. Nous proposons une méthodologie associée à cette démarche. Elle se déroule en quatre étapes principales décrites par la figure 2.1.

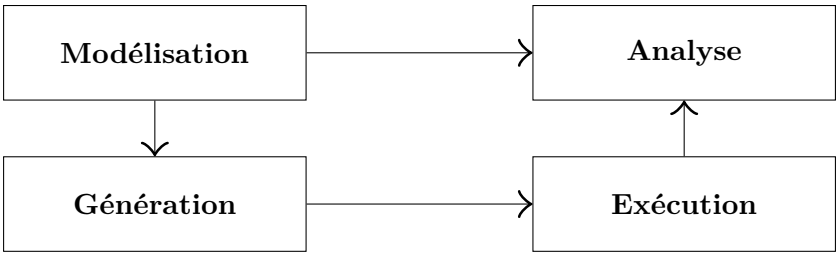


Figure 2.1 – Processus d'analyse du modèle de tâche

La première étape consiste en la définition de l'architecture logicielle à base de composants (Section 2.1). Le modèle d'architecture permet de générer et d'instrumenter du code exécutable par le calculateur du robot (Section 2.2). Le code généré inclut une instrumentation afin de collecter des données d'exécution et ainsi calculer les pires temps d'exécution des composants (Section 2.3). Enfin le modèle d'architecture est combiné avec les temps d'exécution de ses composants afin d'en effectuer l'analyse d'ordonnançabilité en calculant le pire temps de réponse de chaque composants (Section 2.4).

2.1 Architecture à base de composants

Une architecture définit la structure logicielle régissant le comportement global de l'application. Les architectures à base de composants sont des structures logicielles construites à partir de briques fonctionnelles communiquant entre elles. La construction d'architectures à base de composants permet une modularité élevée et offre la possibilité de reconfigurer l'architecture lors de son exécution.

Un composant est une entité logicielle portant une fonctionnalité spécifique de l'ar-

chitecture. Les fonctionnalités implantées au sein des composants sont représentées sous la forme de machines à états. Chaque état contient une section élémentaire de la fonctionnalité et la structure de la machine à états définit le comportement du composant.

Chaque composant est périodiquement exécuté sur le système d'exploitation par des tâches, comme le montre la figure 2.2. Le temps d'exécution de ces tâches dépend de plusieurs facteurs dont la fréquence du processeur, l'exécution d'autres tâches sur le processeur ou des accès bloquants à des périphériques matériels par exemple. Le modèle de composant permet d'associer un temps d'exécution au composant mais aussi à chaque fonction contenue dans les états de la machine à états.

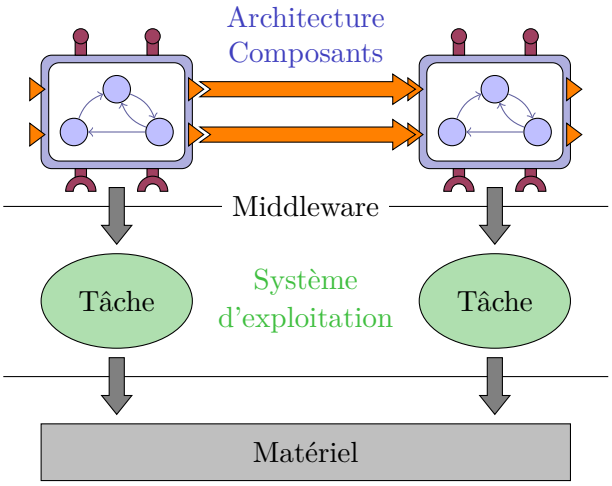


FIGURE 2.2 – Exécution d'une architecture

La modélisation d'architectures à base de composants est présentée dans le chapitre 3.

2.2 Génération de code

Une fois les composants modélisés, l'architecture définie et le déploiement spécifié, nous pouvons générer le code exécutable correspondant. Durant ces travaux, nous avons choisi d'utiliser le middleware Orocos pour ses capacités d'ordonnancement temps réel. Les outils de génération de code sont donc adaptés pour ce middleware.

Le fonctionnement des utilitaires de génération de code consiste à utiliser les modèles de composants et les traduire en code compilable pour le calculateur embarqué du robot. De plus la génération de code permet d'instrumenter le code afin d'intégrer des outils de diagnostic. Cela permet au développeur de se concentrer sur les fonctionnalités essentielles de l'architecture et ainsi réduire les temps de développement.

Le générateur de code est présenté dans le chapitre 7.

2.3 Exécution de l'architecture

L'instrumentation intégrée dans le code généré permet de collecter des informations temporelles sur l'exécution de l'architecture. Chaque composant est observé afin de collecter, pour chaque itération, les dates de début et de fin d'exécution du composant ainsi que de toutes ses fonctions. Une méthode probabiliste d'estimation du temps d'exécution est appliquée aux données datées afin d'obtenir le pire temps d'exécution de chaque composant.

Le chapitre 8 propose un exemple d'architecture développée dans le cadre d'une mission d'exploration autonome par un robot mobile.

2.4 Analyse du système

Enfin, les temps d'exécution des composants sont combinés avec le modèle d'architecture pour calculer le pire temps de réponse de chaque composant. Le pire temps de réponse d'un composant correspond au temps passé entre le début et la fin de son exécution en prenant en compte les éventuelles préemptions des autres composants de l'architecture. L'ordonnançabilité de l'architecture est déterminée si le pire temps de réponse de chaque composant est inférieur à son échéance.

L'analyse de pire temps de réponse est présentée dans le chapitre 6. De plus, ce chapitre fournit une estimation des performances de cette méthode d'analyse en termes de complexité algorithmique et de précision des pires temps de réponse calculés.

Première partie

Modèle de composant, d'architecture et de déploiement

Chapitre 3

Langage de modélisation Mauve

Sommaire

Résumé . *La méthodologie de conception logicielle a évolué ces dernières années pour passer d'un développement orienté vers l'implémentation des fonctionnalités sur une cible matérielle à un développement basé modèles. Ce changement permet un développement plus aisé puisque les fonctionnalités de différentes architectures logicielles peuvent être réutilisées sur diverses plate-formes matérielles. De plus, la modélisation d'architecture logicielle peut être associée à des outils d'analyse et de génération de code permettant respectivement de valider le fonctionnement du logiciel et de réduire les erreurs d'implémentation.*

C'est dans ce cadre que nous avons conçu le langage de modélisation Mauve : il permet de modéliser des architectures logicielles et les composants qui les constituent mais aussi leur déploiement.

Afin de se concentrer sur les fonctionnalités des logiciels et non leur implémentation, nous utilisons le principe de développement basé modèles défini par [Schlegel et Hassler 2009]. Il définit en outre la différence entre les modèles indépendants de la plate-forme permettant de spécifier les fonctionnalités du logiciel et les modèles dépendants de la plate-forme décrivant les spécificités de l'implémentation. Nous nous inscrivons dans une logique de séparation des considérations en découpant le concept d'architecture logicielle en trois éléments [Passama 2006] : les composants qui fournissent des fonctionnalités élémentaires, une architecture qui assemble et lie les composants et un déploiement qui rassemble les informations nécessaires à l'exécution des composants sur la cible matérielle.

Il existe dans l'état de l'art (présenté succinctement dans la section 1.3) un ensemble de langages de modélisation permettant de décrire des modèles de composants, d'architectures et de déploiements mais ils ne supportent pas à la fois l'analyse expliquée au chapitre 6 et la génération de code décrite au chapitre 7 sans de lourdes modifications. Nous avons pour cela conçu notre propre langage de modélisation : le langage Mauve.

Ce chapitre est donc séparé en trois sections : la première décrit la modélisation des composants ainsi que des machines à états associées ; la seconde présente le modèle d'architecture permettant l'instanciation des composants. Enfin la troisième décrit le modèle de déploiement.

Afin de clarifier les illustrations de ce chapitre, nous allons utiliser un exemple d'architecture logicielle inspiré des tutoriels du middleware Orocos [1] :

Exemple 3.1. *Cet exemple consiste à interagir avec un robot mobile. Il est donc constitué d'un robot pouvant être contrôlé soit manuellement par un joystick soit de manière autonome via un script d'exécution. Les deux modes de fonctionnement peuvent être sélectionnés pendant l'exécution. L'architecture est constituée de cinq composants :* plant *qui représente le robot,* joystick *permettant un contrôle manuel,* automatic *qui comporte les scripts d'exécution,* controller *qui ferme la boucle de contrôle de position du robot et enfin* mode_switch *qui gère le passage entre les modes automatique et manuel. L'architecture correspondante est représentée dans la figure 3.1.*

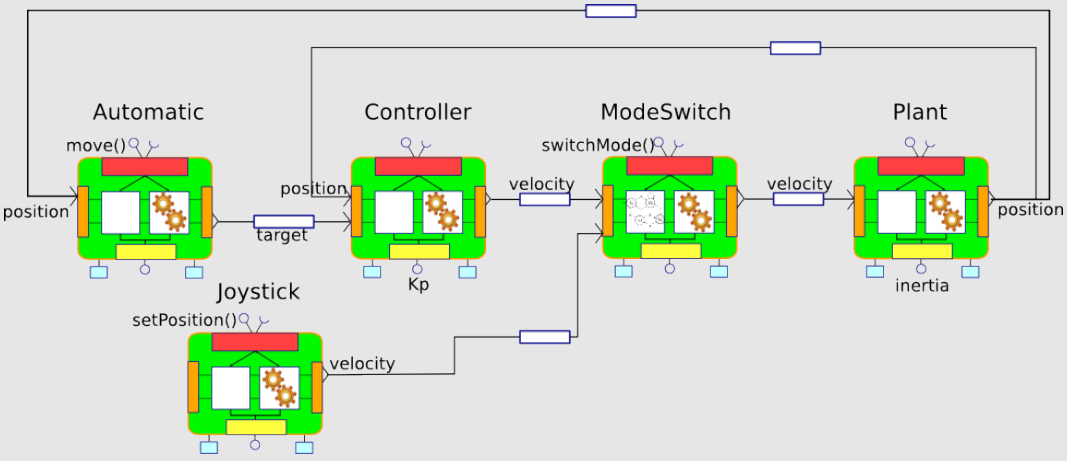


FIGURE 3.1 – Exemple d'architecture logicielle

1. http://www.orocos.org/wiki/orocos/toolchain/getting-started/toolchain-tutorials

3.1 Développement d'un modèle de composant

L'objectif de cette section est de détailler les différents éléments constituants du modèle de composants. Ce modèle de composants a pour but d'intégrer la modélisation de machines à états présentes au sein de certains composants. La modélisation des composants se base sur le langage de modélisation Mauve [Gobillot, Lesire et Doose 2014].

Cette section se découpe en deux parties : nous décrivons tout d'abord la structure d'un composant, constituée d'une interface et de fonctions. Puis nous présentons la modélisation d'architectures et de déploiements permettant l'instanciation de ces composants.

3.1.1 Description d'un composant

Selon [Szyperski, Gruntz et Stephan 202], *un composant est une entité possédant des interfaces spécifiées contractuellement et des dépendances explicites au contexte. Un composant logiciel est sujet à des compositions par des tierces parties mais peut être déployé de manière indépendante.* Par conséquent, afin de conserver la composition et la modularité des composants, nous découpons leur spécification en une *coquille* et un *cœur* définissant respectivement leur interface et leur comportement, comme le montre la figure 3.2.

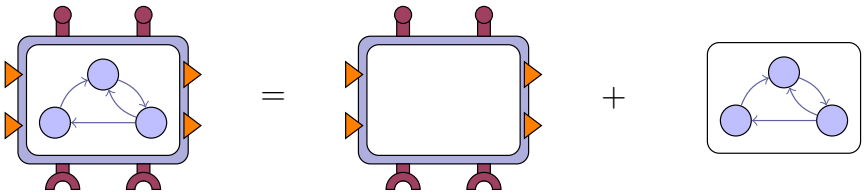


Figure 3.2 – Composant constitué d'une coquille et d'un cœur

3.1.1.1 La coquille d'un composant

La coquille d'un composant définit son interface, c'est à dire ses entrées-sorties, comme le montre la figure 3.3. Nous définissons quatre types d'interfaces : les *constantes*, les *propriétés*, les *ports* et les *opérations*.

— les constantes : sont des valeurs statiques et typées spécifiques à une coquille. Les constantes sont définies et leur valeur est attribuée lors de la spécification de la coquille.

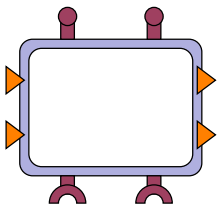


FIGURE 3.3 – Coquille d'un composant

— les propriétés : sont les paramètres typés des composants. En pratique les valeurs des propriétés sont souvent attribuées lors de l'instanciation ou du déploiement mais peuvent avoir une valeur par défaut.
— les ports : similaires à la forme *push* proposée par [SCHLEGEL 2006], ils permettent de publier ou de recevoir des données vers ou depuis d'autres composants. Les ports sont typés et directionnels.
— les opérations : similaires à la forme *query* proposée par [SCHLEGEL 2006], ils sont utilisés pour appeler des fonctions ou envoyer des requêtes à d'autres composants.

La spécification d'une coquille de composant s'effectue donc en listant l'ensemble de ses constantes, de ses propriétés, de ses ports et de ses opérations. Cependant les valeurs des propriétés ainsi que les connexions des ports et des opérations ne sont pas attribuées lors de la spécification de la coquille mais plutôt lors de l'instanciation des composants, au moment de la spécification de l'architecture (voir la section 3.2).

Exemple 3.2. *Le listing 3.1 montre la spécification de la coquille du composant* plant. *La coquille* `PlantShell` *possède une propriété de type* virgule flottante *(double) appelée* `inertia`. *Cette coquille possède aussi deux ports,* `velocity` *et* `position`, *l'un en entrée et l'autre en sortie, tous deux de type* virgule flottante.

Listing 3.1 – Coquille du composant *plant*

```
1 shell PlantShell {
2   input port velocity: double
3   output port position: double
4   property inertia: double
5 }
```

3.1.1.2 Le cœur d'un composant

Le cœur d'un composant définit son comportement et doit être associé à une coquille, comme le montre la figure 3.4. Le comportement des composants peut être défini de deux manières différentes : soit en exécutant périodiquement une fonction, soit en spécifiant une machine à états. La spécification des machines à états est présentée dans la section 3.1.2

Le cœur d'un composant est constitué de trois types d'éléments : des *variables*, des

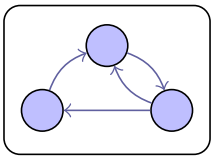


FIGURE 3.4 – Cœur d'un composant

opérations et des *programmes*.

- les variables : sont des éléments internes au composant permettant de stocker des valeurs modifiables utilisées dans le langage d'expression Mauve.
- les opérations : ont été déclarées dans la coquille. Leur implémentation est effectuée au sein du cœur du composant.
- les programmes : il existe cinq types de programmes différents qui contiennent les fonctions, aussi appelées *codels* [MALLET, PASTEUR, HERRB et al. 2010], du composant : `configure`, `start`, `update`, `stop` et `cleanup`. Lorsqu'une machine à états est spécifiée, le programme `update` est remplacé par `statemachine`.
  - `configure` : ce programme contient les fonctions appelées lors de l'initialisation du composant, au moment de l'exécution. `configure` retourne une valeur booléenne mise à *vrai* si la configuration s'est effectuée correctement et à *faux* sinon.
  - `start` : ce programme ne peut être appelé que lorsque la configuration a été effectuée. Il est appelé afin que le composant commence son exécution périodique. `start` retourne une valeur booléenne mise à *vrai* si le démarrage s'est effectué correctement et à *faux* sinon.
  - `update/statemachine` : ce programme s'exécute à chaque cycle d'exécution du composant s'il a été démarré avec `start`.
  - `stop` : ce programme est exécuté afin d'arrêter le composant.
  - `cleanup` : ce programme sert à nettoyer le composant lorsqu'il est déchargé du système.

Les différents programmes ne peuvent être exécutés que dans un ordre précis présenté dans la figure 3.5 : après avoir chargé le composant, il faut l'initialiser avec `configure`

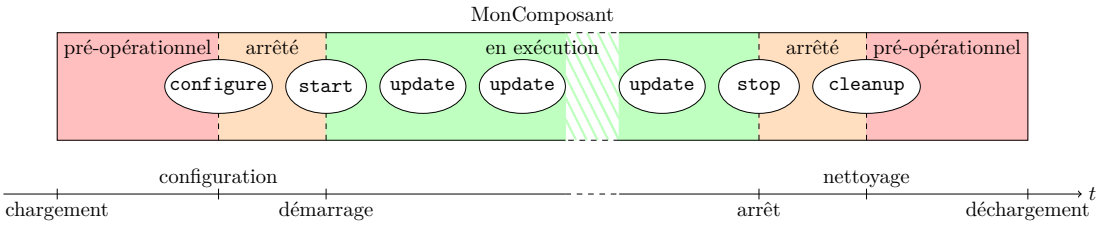


FIGURE 3.5 – Séquence d'exécution des programmes dans un cœur de composant

puis le démarrer avec `start`. Suite à l'appel de ces deux programmes, le composant

exécute périodiquement `update` ou `statemachine` jusqu'à l'appel de `stop`. Enfin, le composant est nettoyé avec `cleanup` pour pouvoir le décharger de l'architecture.

Exemple 3.3. *Le listing 3.2 montre la spécification du cœur du composant* plant. *Le cœur* `PlantCore` *est associé dès sa déclaration à la coquille* `PlantShell` *afin de lui apporter son interface. Ce cœur possède deux variables* `pos` *et* `cmd` *dont seule la seconde est initialisée à 0.*

Parmi les cinq programmes `configure`, `start`, `stop`, `cleanup` *et* `update`, *seuls* `start` *et* `update` *sont définis avec des appels aux fonctions du composant et à son interface. Le programme* `start` *est utilisé pour initialiser la valeur de la variable* `pos` *à 0 lors du démarrage du composant. Le programme* `update`, *exécuté à chaque période du composant, effectue trois actions : il commence par lire le port d'entrée* `velocity` *et stocke la valeur dans la variable* `cmd`, *puis il met à jour la valeur de son état* `pos` *grâce au codel* `compute_position`. *Enfin il écrit son état sur le port* `position`.

Listing 3.2 – Cœur du composant *plant*

```
1 core PlantCore (PlantShell) {
2   var pos: double;
3   var cmd: double = 0.0;
4
5   start = { pos = 0.0; return true; }
6
7   update = {
8     read(velocity, cmd);
9     pos = compute_position(pos, cmd, inertia, (period/1000.0));
10    write(position, pos);
11  }
12 }
```

3.1.2 Description d'une machine à états

Chaque composant peut contenir une machine à états qui remplace le programme `update`. Une machine à états consiste en un ensemble d'états reliés par un ensemble de transitions (voir la définition 4.1).

La structure des machines à états Mauve est similaire aux *rFSM* [Klotzbücher et Bruyninckx 2012], notamment utilisées par le middleware Orocos, elles mêmes dérivées des *Statecharts* UML [Varró 2002].

Chaque état s_i contient jusqu'à quatre méthodes : $entry_i$, run_i, $handle_i$ et $exit_i$. La méthode $entry_i$ contient le code exécuté lorsque la machine à états entre dans l'état s_i. La méthode run_i contient le cœur de l'état, exécuté à chaque fois que la machine à états est dans l'état s_i. La méthode $handle_i$ est exécutée à chaque fois que la machine à états

reste dans le même état, juste après run_i. Enfin, la méthode $exit_i$ est exécutée lorsque la machine à états quitte l'état s_i.

Exemple 3.4. *Le listing 3.3 décrit la spécification de la machine à états du composant* mode_switch. *Les machines à états ne sont pas nommées puisqu'elles sont uniques par composant. Dans cet exemple, la machine à états possède trois états, comme présenté dans la figure 3.6 :*

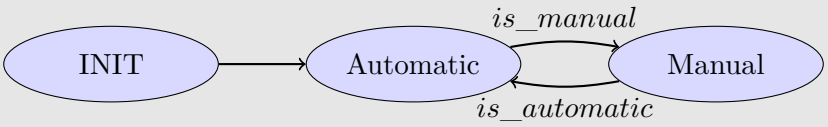


FIGURE 3.6 – Une machine à états périodique simplifiée

- *l'état initial* `INIT` *qui initialise la variable* `mode` *à* `automatic_mode` *lorsque la machine à états entre dans cet état et effectue une transition vers l'état* `Automatic`.
- *un second état, appelé* `Automatic` *qui lit le port* `auto_vel` *pour stocker sa valeur dans la variable* `command`. *Ensuite, si la fonction* `is_manual(mode)` *renvoie la valeur* `true`, *la machine à états effectue une transition vers l'état* `Manual`. *Sinon, l'état écrit le contenu de* `command` *dans le port* `velocity`.
- *le troisième état,* `Manual` *est le pendant de* `Automatic` *mais au lieu de lire le port* `auto_vel`, *il utilise le port* `manual_vel` *pour transférer sa valeur à* `velocity`.

Listing 3.3 – Machine à états du composant *mode_switch*

```
1  statemachine {
2    initial state INIT {
3      entry {
4        mode = automatic_mode;
5      }
6      transition select Automatic
7    }
8    state Automatic {
9      run {
10       read(auto_vel, command);
11     }
12     handle {
13       write(velocity, command);
14     }
15     transition if (is_manual(mode)) select Manual
16   }
17   state Manual {
18     run {
19       read(manual_vel, command);
20     }
21     handle {
22       write(velocity, command);
```

```
23      }
24      transition if (is_automatic(mode)) select Automatic
25    }
26  }
```

3.2 Développement d'un modèle d'architecture

La conception de la couche logicielle fonctionnelle pour des robots autonomes se base sur la réutilisation et sur l'association de composants de base. La conception d'une *architecture* logicielle consiste donc en l'instanciation de composants définis dans la section 3.1 ainsi qu'en la définition de propriétés. Les composants sont ensuite connectés entre eux en spécifiant la configuration du protocole de communication, comme par exemple la présence ou non d'une mémoire tampon ainsi que la spécification de la politique de gestion du tampon.

Exemple 3.5. *La figure 3.7 présente une partie de l'architecture présentée dans l'exemple 3.1 : la partie de contrôle automatique.*

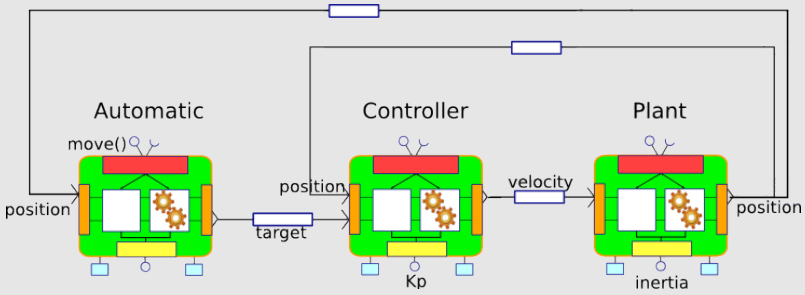


FIGURE 3.7 – Sous partie de l'architecture logicielle

Le listing 3.4 montre la spécification de cette partie de l'architecture. Les composants `plant`, `controller` *et* `automatic` *sont instanciés puis connectés.*

Listing 3.4 – Partie automatique de l'architecture de contrôle du robot

```
1  architecture AutomaticControlArchitecture {
2
3    instance plant: Plant
4    instance controller: Controller
5    instance automatic: Automatic
6
7    connection automatic.command -> controller.target
8    connection controller.velocity -> plant.velocity
9    connection plant.position -> controler.position
10   connection plant.position -> automatic.position
11 }
```

3.3 Développement d'un modèle de déploiement

Le déploiement est la partie du développement logiciel spécifique à la cible matérielle sur laquelle s'exécute l'architecture. Il permet d'associer les composants et la spécification de l'architecture aux deux éléments principaux de la plate-forme, à savoir le *matériel* et le *middleware*.

Le matériel, aussi appelé la cible, correspond à la plate-forme réelle utilisée lors des expérimentations. Il consiste en l'ensemble des capteurs, actionneurs et calculateurs présents sur le robot. Dans ce but, le langage d'expression Mauve permet de spécifier les propriétés des composants de l'architecture, comme les ports sur lesquels sont connectés le matériel ou le type de processeur utilisé.

Le middleware consiste en une couche additionnelle sur le système d'exploitation installé sur les calculateurs. Il offre des fonctionnalités facilitant le développement d'architectures logicielles en abstrayant les spécificités du système d'exploitation. Mauve ne supporte à l'heure actuelle que le middleware Orocos et ne propose donc pas de moyen de choisir le middleware.

Lors du déploiement des composants, une activité leur est attachée afin de spécifier leur paramètres d'exécution : la période, l'échéance, la priorité et l'affinité. La période d'un composant indique sa fréquence d'activation. L'échéance représente quant à elle la date au plus tard à laquelle le composant doit avoir terminé son exécution. La priorité est utilisée par l'ordonnanceur afin de décider si un composant peut en préempter un autre. L'affinité spécifie, dans un contexte d'exécution multi-cœur, sur quel cœur du processeur doit s'exécuter le composant. Ces paramètres sont ensuite associés aux activités des composants Orocos puis aux propriétés des processus correspondants.

Exemple 3.6. *Le listing 3.5 montre le déploiement de la partie automatique de l'architecture de contrôle. Pour cela, l'architecture est tout d'abord instanciée puis les propriétés des composants dépendantes de l'implémentation sont initialisées. Enfin chaque composant est associé à une tâche et les paramètres de ces tâches sont définis, à savoir : l'affinité, la priorité, la période et l'échéance.*

Listing 3.5 – Déploiement correspondant à la partie automatique de l'architecture

```
1 deployment AutomaticControlDeployment {
2   architecture AutomaticControlArchitecture
3
4   property plant.inertia = 1.0
5   property controller.Kp = 0.5
```

```
6
7   activity plant {
8     affinity = 1
9     priority = 1
10    period   = 100
11    deadline = 100
12  }
13
14  activity controller {
15    affinity = 1
16    priority = 2
17    period   = 100
18    deadline = 100
19  }
20
21  activity automatic {
22    affinity = 1
23    priority = 3
24    period   = 1000
25    deadline = 1000
26  }
27 }
```

3.4 Conclusion

Nous avons développé un langage de modélisation permettant de représenter les trois éléments principaux d'une architecture logicielle pour la robotique : les composants, les architectures et les déploiements. Ils sont organisés sous la forme de librairies afin de faciliter la réutilisation d'éléments au sein d'autres librairies.

Les composants constituent le premier élément modélisé par Mauve. Ils sont constitués de deux éléments : une *coquille* qui fournit l'interface du composant et un *cœur* qui, associé à une coquille, définit son comportement. Le comportement du composant peut être spécifié sous la forme d'une fonction unique ou sous la forme d'une machine à états. Le deuxième élément modélisé est décrit par les architectures. Une architecture permet d'assembler, d'instancier et de configurer les composants précédemment définis. De plus, les architectures définissent les connexions entre composants ainsi que les politiques de connexion. Enfin, le troisième élément modélisé par Mauve est constitué par les déploiements. Un déploiement permet d'instancier une architecture et de configurer les paramètres d'exécution des composants. L'ensemble de ces éléments constitue différents niveaux de granularité permettant d'augmenter la réutilisation du code : une même coquille peut être utilisée pour plusieurs cœurs de composants ; un composant peut être utilisé pour plusieurs architectures et une architecture peut être utilisée pour plusieurs déploiements.

La grammaire détaillée du langage de modélisation Mauve est disponible dans l'an-

nexe A. La grammaire est amenée à évoluer et sa description est disponible à l'adresse suivante : https://forge.onera.fr/projects/mauve/wiki/Language_definition.

L'étape suivante consiste à associer le modèle de composant à un modèle de tâche. Cela permet de lier la représentation fonctionnelle des modèles Mauve à une représentation temporelle. Cette-ci nous permettra par la suite d'obtenir les temps d'exécution des différents codels pour finalement étudier le respect des échéances des composants.

Chapitre 4

Développement d'un modèle d'exécution de tâches

Sommaire

Résumé . *La modélisation de tâches et d'architectures permet de définir une structure logicielle analysable mais n'indique pas de quelle manière les composants constituants les architectures s'exécutent. De plus, l'exécution réelle de tâches sur un calculateur s'effectue en un temps non nul. Ce chapitre décrit donc le modèle d'exécution des composants sous la forme de tâches, la sémantique d'exécution des machines à états ainsi que la méthode d'obtention des temps d'exécution des éléments constitutifs des tâches.*

Une tâche est une entité logicielle portant une ou plusieurs fonctionnalités représentant leur comportement fonctionnel. Nous utilisons une machine à états au sein de chaque tâche afin de décrire le comportement et la séquentialité des fonctions internes aux tâches.

La modélisation des composants à été décrite dans le chapitre précédent. Cependant, ces composants doivent être associés à des tâches du système d'exploitation pour pouvoir s'exécuter. Il est donc nécessaire de fournir un modèle temporel d'exécution pour ces tâches.

Ce chapitre est découpé en trois parties : la première définit le modèle de tâches utilisé lors du déploiement des composants. Puis la seconde partie définit la modélisation

des machines à états du point de vue de l'exécution. Enfin la troisième section décrit la méthode d'obtention des temps d'exécution des composants.

4.1 Description du modèle d'exécution de tâches

Le modèle de tâche permet de décrire deux éléments : le premier correspond à l'impact de la définition des composants fonctionnels sur leur exécution. Le second élément décrit le comportement temporel des tâches.

4.1.1 Indépendance entre tâches

Les composants de l'architecture sont conçus pour être temporellement indépendants. Les deux implications principales de cette indépendance temporelle sont : les composants doivent pouvoir s'exécuter sans contraintes et les échanges de données entre composants ne doivent pas interrompre leur exécution.

La conception des composants s'effectue sous la forme d'entités temporellement indépendantes. Pour que leur exécution soit conforme à cette spécification, leur implémentation est effectuée sous la forme de tâches indépendantes : l'activation d'une tâche n'est donc pas liée à l'exécution d'une autre.

Hypothèse 4.1. *Chaque composant est associé à une et une seule tâche lors du déploiement.*

Les composants d'une architecture logicielle échangent des données par l'intermédiaire de ports d'entrée/sortie. Pour éviter l'attente de données par les composants, nous utilisons un mécanisme de communication asynchrone de type publication/souscription. Ce mécanisme permet à un composant de recevoir une donnée venant d'un autre composant sans avoir à attendre que celle-ci soit disponible. La fraicheur des données est indiquée lors de la lecture de celle-ci.

Hypothèse 4.2. *Les communications entre tâches sont non-bloquantes.*

4.1.2 Modèle de tâche

Les composants sont liés à des tâches du système d'exploitation par le middleware. Le comportement des tâches est donc défini par celui du composant et hérite donc de son activité.

Comme présenté dans la section 1.4 de l'état de l'art, il existe différents modèles de tâches selon les conditions d'exécution et la précision attendue. Plus un modèle est précis, plus les analyses d'ordonnancement les utilisant sont précises mais aussi plus le temps nécessaire afin d'effectuer ces analyses est long. Nous avons choisi pour ces travaux d'étendre le modèle canonique de tâches pour augmenter son expressivité et sa précision tout en maintenant une complexité d'analyse faible (voir la section 6.4.1).

Une tâche τ_i est définie par quatre éléments : la période, la priorité, l'échéance et un comportement.

- La période T_i permet de configurer la fréquence d'exécution des tâches ;
- l'échéance D_i correspond à la date à laquelle les tâches doivent terminer leur exécution ;
- la priorité P_i est gérée par l'ordonnanceur à partir, entre autres, de la période et de l'échéance ;
- le comportement des tâches est défini par la modélisation de la structure interne de la tâche SM_i pouvant représenter soit une machine à états, soit une fonction unique et indivisible. Le comportement des tâches permet d'en déduire le temps d'exécution en fonction de la cible matérielle et du déroulement de la machine à états.

La représentation du comportement sous la forme d'une machine à états est préférée puisqu'elle apporte plus de précision à la modélisation que la représentation sous la forme d'une fonction.

La tâche τ_i a une exécution périodique de période T_i. L'hypothèse suivante est supposée vraie pour la suite du développement :

Hypothèse 4.3. *Les échéances sont inférieures ou égales aux périodes :*

$$\forall i, D_i \leq T_i \tag{4.1}$$

De plus, le choix initial du middleware Orocos ne permet pas de synchroniser ni de connaître les dates de réveil des tâches.

Hypothèse 4.4. *Les dates de réveil des tâches sont inconnues lors du déploiement.*

4.1.3 Description d'une machine à états

Le comportement temporel d'une tâche dépend à la fois des propriétés temps-réel des activités de la tâches ainsi que de son comportement interne lié à sa machine à états.

Définition 4.1. *Une machine à états est définie par un ensemble S de n états et un ensemble E de m transitions tel que :*

$$S = \{s_1, \ldots, s_n\} \text{ et } |S| = n \tag{4.2}$$
$$E = \{e_1, \ldots, e_m\} \text{ et } |E| = m \tag{4.3}$$

Les machines à états portent toutes les fonctions et algorithmes des tâches. Ces fonctions utilisent du temps processeur et ce modèle de tâche permet d'utiliser la structure des machines à états pour déterminer le pire temps de réponse des tâches.

Pour une machine à états SM_i, deux cycles d'exécution sont possibles par itération :
— la machine à états reste dans le même état s_i pour l'itération suivante : les méthodes run_i et $handle_i$ sont exécutées ;
— la machine à états passe de l'état s_i à l'état s_j : les méthodes run_i, $exit_i$ puis $entry_j$ sont exécutées.

Les quatre méthodes n'ont pas besoin d'être définies pour chaque état selon l'application mais run_i doit l'être.

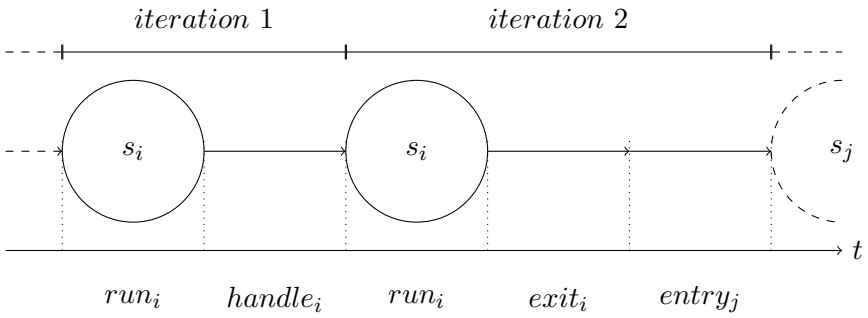


FIGURE 4.1 – Protocole de transition d'une machine à états

Deux itérations successives de l'exécution d'une machine à états sont montrées sur la figure 4.1. Cette machine à états contient les méthodes $entry_i$, run_i, $handle_i$ et $exit_j$. Lors de la première itération, la machine à états reste dans l'état s_i et exécute donc les méthodes run_i et $handle_i$. Lors du passage de l'état s_i à l'état s_j de la seconde itération, les méthodes run_i, $exit_i$ et $entry_j$ sont exécutées.

4.2 Calcul du pire temps d'exécution des tâches

Les tâches exécutant les fonctions et les machines à états des composants utilisent les ressources de calcul fournies par le processeur. Le temps passé par le processeur à calculer

une fonction est appelé *temps d'exécution*. Cependant le temps d'exécution d'une fonction varie selon les conditions dans lesquelles elle s'exécute. Le *pire temps d'exécution* d'une fonction est défini comme une borne supérieure du temps d'exécution : il correspond au temps passé par le processeur à calculer cette fonction dans les conditions les plus défavorables à l'exécution.

Afin d'exploiter la précision apportée par les machines à états, nous devons déterminer les pires temps d'exécution des fonctions appelées par les machines à états, et plus précisément les pires temps d'exécution des méthodes *entry*, *run*, *handle* et *exit* des états des machines à états périodiques. Pour cela, nous utilisons deux méthodes : soit une analyse statique, soit une analyse probabiliste.

4.2.1 Analyse statique

Les méthodes d'analyse statique [Wilhelm et al. 2008] calculent de manière déterministe le pire temps d'exécution de fonctions à partir de deux éléments : un modèle du processeur sur lequel va s'exécuter la fonction et un exécutable compilé de la fonction.

Au cours de nos expérimentations, nous avons utilisé l'outil Otawa [Ballabriga et al. 2010] pour collecter les pires temps d'exécution d'une architecture robotique similaire à celle présentée dans le chapitre 8. Le principe de fonctionnement de Otawa se base sur deux éléments principaux : des abstractions et des analyses. Les niveaux d'abstraction permettent de formaliser les spécifications de l'architecture matérielle du processeur cible et de fournir une représentation structurelle du programme exécuté sur ce processeur. Les méthodes d'analyse collectent les informations fournies par les couches d'abstraction afin de déterminer le pire temps d'exécution du programme sur le processeur cible.

Dans ces expérimentations, nous avons utilisé des cartes de développement Gumstix® Overo® Water à base de processeur ARM® Cortex™A8. Le modèle de processeur de cette carte de développement est disponible pour les outils d'abstraction matérielle de Otawa. Les modèles de processeur contiennent des informations détaillées des trois éléments principaux, à savoir le calculateur en lui même, les différents niveaux de cache et la mémoire principale :

— le modèle du calculateur fournit des informations sur la taille du *pipeline*, le nombre d'unités de calcul et leur latences, les catégories d'instructions associées aux unités de calcul ou encore les spécifications des files d'instruction ;
— le modèle de cache indique sa capacité, son organisation et ses diverses politiques de remplacement ou d'écriture ;
— le modèle de mémoire permet de représenter les temps d'accès aux données ainsi que le fonctionnement de la mémoire tampon.

Pour déterminer le pire temps d'exécution d'un programme, les méthodes d'analyse nécessitent une représentation binaire du programme. Otawa effectue en premier lieu une recherche de boucles au sein du programme et essaye de déterminer une borne supérieure du nombre d'itérations de ces boucles. Afin de simplifier cette recherche, le code du programme peut être annoté en indiquant directement le nombre maximal d'itérations des boucles. Ensuite l'exécution du programme est simulée sur le modèle du processeur pour déterminer l'utilisation des caches, les contraintes liées à la taille des caches et une estimation du temps de calcul de blocs d'instructions. Enfin ces informations sont regroupées pour calculer le pire temps d'exécution du programme complet.

4.2.2 Analyse probabiliste

Les variations des temps d'exécution réels sont liées à la variabilité intrinsèque du matériel [Borkar 2005], à la complexité des fonctions [Burns et Littlewood 2010] et des interférences entre les éléments logiciels impliquant des dépendances complexes.

Les techniques d'analyse probabiliste à base de mesures permettent d'estimer le pire temps d'exécution de fonctions à partir de mesures d'exécutions réelles [Cucu-Grosjean et al. 2012 ; Hansen, Hissam et Moreno 2009]. L'utilisation de méthodes probabilistes induit la notion de pire temps de réponse probabiliste $\overline{\mathcal{X}}$ qui représente une distribution de pire temps d'exécution.

La distribution de pire temps d'exécution probabiliste $\overline{\mathcal{X}}$ d'une fonction correspond à la borne supérieure du temps d'exécution $\mathcal{X}_i$ dans toutes les conditions possibles d'exécution i. $\mathcal{X}_i$ est une distribution empirique des temps d'exécution mesurés dans des conditions spécifiques. Par conséquent, pour toute condition i, la distribution $\overline{\mathcal{X}}$ est supérieure ou égale au temps d'exécution $\mathcal{X}_i$. Le pire temps d'exécution probabiliste $\overline{\mathcal{X}}$ est toujours supérieur ou égal à tous les temps d'exécution possibles par le système.

Les mesures $\mathcal{X}_i$ permettent d'extraire des comportements moyens ou des tendances lors de l'exécution des fonctions. Cependant, les mesures de temps d'exécution ne sont pas suffisantes pour obtenir le pire temps d'exécution probabiliste : une exécution réelle d'une fonction ne garantit pas que toutes les conditions possibles d'exécution puissent être observées.

Afin de palier à ce manque de complétude, nous utilisons la théorie des valeurs extrêmes [Santinelli et al. 2014]. Cela permet d'obtenir une borne non optimiste de l'estimation du pire temps d'exécution probabiliste $\overline{\mathcal{X}}$, c'est à dire que $\overline{\mathcal{X}}$ est supérieur ou égal à $\mathcal{X}$. La théorie des valeurs extrêmes permet de « prédire » le pire temps d'exécution, même si celui-ci n'est pas observable lors d'une expérimentation. La figure 4.2 montre une distribution de probabilité d'occurrence de temps d'exécution mesurés ainsi que la

distribution *prédite* par la théorie des valeurs extrêmes. La prédiction calcule un pire temps d'exécution sûr supérieur au pire temps exact.

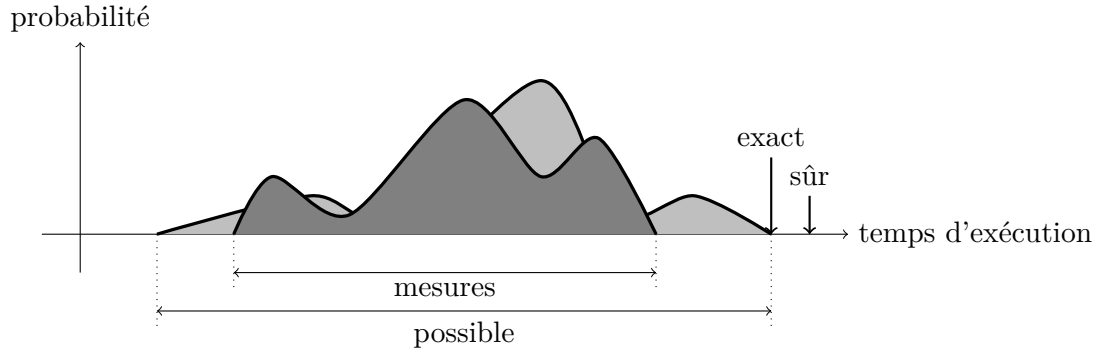


FIGURE 4.2 – Comparaison entre les temps d'exécution mesurés et estimés par la théorie des valeurs extrêmes

Ensuite nous estimons les pires temps d'exécution $\mathcal{C}$ à partir des pires temps d'exécution probabilistes $\overline{\mathcal{X}}$ des fonctions et en les limitant à une borne de confiance p. Le couple $\langle \overline{\mathcal{X}}, p \rangle$ représente l'estimation du pire temps d'exécution pour une fonction donnée.

En pratique, les mesures nécessaires au calcul de pire temps d'exécution probabiliste sont collectées par l'intermédiaire d'un traceur. Ce traceur enregistre les dates d'exécution d'un ensemble de points dans le code des composants. Pour chaque composant, le générateur de code présenté dans le chapitre 7 instrumente le code exécutable en différents endroits. Un point d'enregistrement est ajouté autour de chaque élément des composants :

— autour du `core` du composant afin de mesurer le délai entre le début et la fin d'exécution du composant, en prenant en compte les délais induits par le *middleware* ;
— autour du `update` ou de la machine à états pour collecter le temps d'exécution du composant sans prendre en compte le *middlware* ;
— autour de chaque fonctions et de chaque état de la machine à états dans le but de mesurer le temps nécessaire au calcul des codels.

Pour cela nous utilisons le traceur *Linux Trace Toolkit : next generation* [BROWN 2006], plus communément appelé LTTng, pour enregistrer des mesures d'exécution, comme présenté dans l'exemple 4.1. Les mesures collectées concernent à la fois les fonctions des composants ainsi que les tâches auxquelles ils sont attachés. Dans les deux cas, les mesures permettent d'obtenir la durée réelle d'exécution ce ces éléments. Chaque mesure est constituée de plusieurs éléments :

— une date absolue en secondes ;
— la différence entre la date courante et la date précédente en secondes ;
— le nom de la session utilisateur ;

— le type de mesure. Dans le cas de Mauve, deux types de mesures existent : `Mauve:component:` et `Mauve:codel:` qui représentent respectivement une mesure concernant un composant et un codel ;
— le numéro du cœur de processeur sur lequel s'est exécutée la mesure ;
— des informations dépendantes du type de mesure. Pour une mesure de type `Mauve:component:`, ces informations sont : l'identifiant de la tâche exécutant le composant (`thread_id`), la position de la mesure (`state`) pouvant être soit au début (`begin`) soit à la fin (`end`) de l'exécution du composant et le nom du composant (`component`). Pour une mesure de type `Mauve:codel:`, les informations contenues dans la mesure sont : l'identifiant de la tâche exécutant le composant (`thread_id`), la position de la mesure (`state`) et le nom du codel exécuté (`codel`).

Exemple 4.1. *Afin d'illustrer les données récoltées avec le traceur LTTng, nous avons enregistré l'exécution de l'architecture présentée dans le chapitre 8. Cette architecture est constituée de plusieurs composants, dont un composant de cartographie et de localisation simultanée (SLAM).*

Les mesures fournies par LTTng sont présentées dans le listing 4.1.

Listing 4.1 – Log partiel d'exécution d'une architecture logicielle pour la robotique

```
1 [1424868661.227264005] (+0.000782772) miranda Mauve:component: { cpu_id =
      0 }, { thread_id = 13855, state = "begin", component = "slam" }
2 [1424868661.227298881] (+0.000034876) miranda Mauve:codel: { cpu_id = 0 },
       { thread_id = 13855, state = "begin", codel = "addScan" }
3 [1424868661.227353428] (+0.000054547) miranda Mauve:codel: { cpu_id = 0 },
       { thread_id = 13855, state = "end", codel = "addScan" }
```

La première ligne de ce listing représente la mesure du début d'exécution d'une instance du composant GMapping, ici appelé ***slam****. Les deux autres correspondent respectivement au début et à la fin d'exécution du codel* ***addScan*** *du composant* ***slam****.*

Ces mesures brutes représentent les temps de réponse réels des codels et des tâches. Afin d'obtenir les temps d'exécution, les mesures sont traitées en extrayant toutes les préemptions provenant des autres éléments de l'architecture logicielle. Cela permet de tracer un histogramme représentant la probabilité d'occurrence des temps d'exécutions, comme le montre la figure 4.3 pour le codel `addScan` du composant de cartographie GMapping.

Enfin le pire temps d'exécution probabiliste est extrait à partir des distributions de temps d'exécution de chaque codel. Pour cela, nous calculons une distribution cumulée inverse de temps d'exécution à partir des mesures. Puis nous effectuons un ensemble de tests permettant de déterminer la stationnarité et l'indépendance des données pour vérifier l'applicabilité de la théorie des valeurs extrêmes. Dans le cas où la théorie des valeurs extrêmes est applicable, elle fournit une estimation du pire temps d'exécution

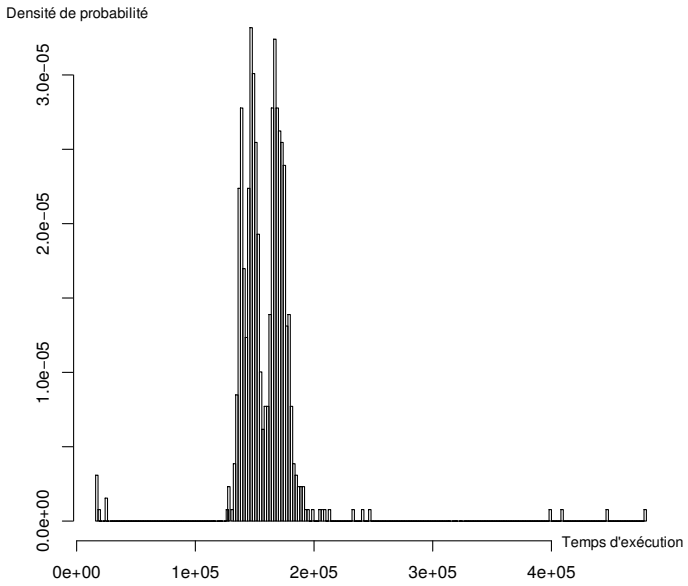


FIGURE 4.3 – Histogramme des temps d'exécution mesurés du codel `addScan` du composant de cartographie GMapping

probabiliste pour un certain indice de confiance. D'un point de vue pratique, nous avons constaté l'applicabilité de la théorie des valeurs extrêmes pour des ensemble de données expérimentales de quelques milliers de point par codel. La figure 4.4 montre la distribution cumulée inverse des temps d'exécution du codel `addScan`. Avec une confiance de 10^{-7}, l'estimation du pire temps d'exécution de cette fonction est de 1 379 ms.

Comme les traces obtenues à partir des exécutions ne considèrent pas seulement les fonctions des composants mais aussi le reste du composant, il est possible d'estimer le temps passé hors des fonctions implémentées, appelée la *glue*. Le pire temps d'exécution de la glue correspond au temps passé à lire ou écrire dans les ports de données, au temps passé par le middleware pour déclencher les fonctions des machines à états ou encore au temps pris par le système d'exploitation pour réveiller les tâches associées aux composants. De ce fait, l'approche à base de mesures de temps d'exécution permet de borner l'influence du système sur le comportement des composants.

4.3 Conclusion

L'objectif de ce chapitre est de définir le modèle de tâches ainsi que de décrire les méthodes d'obtention des temps d'exécution des codels contenus au sein des composants.

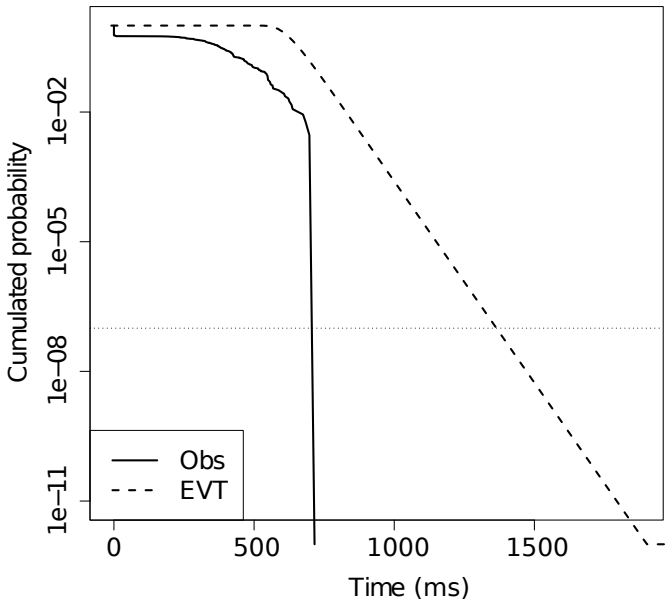


FIGURE 4.4 – Distribution cumulée inverse des temps d'exécution de la fonction `addScan`. Obs est la fréquence des temps d'exécution mesurés $\mathcal{X}$, EVT est l'estimation probabiliste du pire temps d'exécution $\overline{\mathcal{X}}$.

Le modèle de tâche associé à Mauve possède une structure similaire au modèle canonique de tâche. La différence principale concerne le temps d'exécution des tâches : il est déterminé à partir de la structure des machines à états.

Les temps d'exécution des codels sont extraits selon deux techniques différentes : soit par analyse statique de code, soit par analyse probabiliste. Cependant, ces deux techniques possèdent des limites à différents niveaux les rendant complémentaires. L'analyse statique fournit un pire temps d'exécution exact lorsque le modèle de processeur est disponible et que le code respecte des critères permettant son analyse, comme par exemple le bornage des boucles. Lorsque ces critères ne sont pas clairement identifiables au sein du code, il faut l'annoter afin de fournir les informations manquantes. L'analyse probabiliste donne quant à elle un pire temps d'exécution sûr à partir de mesures effectuées lors d'exécutions réelles, quelque soit le type de processeur. Cependant, les mesures effectuées pour déterminer le pire temps d'exécution de fonctions ne sont pas forcément représentatives des cas d'application réels. Ces deux techniques de calcul de pire temps d'exécution sont donc complémentaires. Dans la pratique, nous avons privilégié l'utilisation de temps d'exécutions probabilistes puisque notre robot est équipé d'un calculateur possédant un processeur Intel® Celeron® dont le modèle n'est pas disponible pour les analyses statiques.

L'étape suivante consiste à utiliser les modèles de composants Mauve ainsi que les données de pire temps d'exécution des codels pour effectuer une analyse d'ordonnançabilité.

Deuxième partie

Analyse d'ordonnançabilité utilisant des machines à états

Chapitre 5

Protocole d'analyse

Sommaire

Cette partie présente la méthode d'analyse d'architectures logicielles à base de composants contenant des machines à états. La méthodologie associée à cette démarche se déroule en quatre étapes principales décrites par la figure 5.1. Chaque composant est

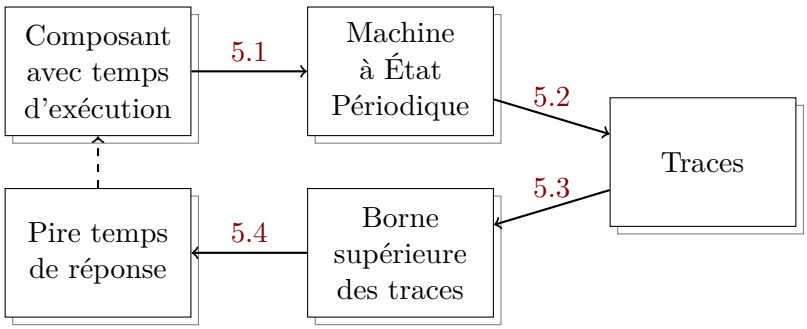


Figure 5.1 – Processus d'analyse du modèle de tâche

défini à partir de machines à états dont les états possèdent des temps d'exécution. La première étape de l'analyse consiste à modéliser les composants sous la forme de machines à états périodiques. Une machine à états périodique correspond à une machine à états dont les fonctions sont portées par les transitions (Section 5.1). La représentation des composants sous la forme de machines à états périodiques permet d'extraire un ensemble de traces d'exécution sous la forme de fonctions de requête (Section 5.2). L'étape suivante consiste à obtenir une borne supérieure de l'ensemble de traces pour chaque composant (Section 5.3). Enfin les pires temps de réponse de chaque composant sont calculés à partir des bornes supérieures des traces des composants de priorité plus élevée afin de décider de l'ordonnançabilité de l'architecture (Section 5.4).

5.1 Machines à États Périodiques

La représentation sous forme de machines à états de la fonctionnalité des composants permet d'exprimer le comportement fonctionnel mais pas temporel. Dans le but d'effectuer une analyse temporelle sur les composants comportant des machines à états, nous introduisons une simplification de leur modèle : les *machines à états périodiques*. Une machine à états périodique correspond à une machine à états dont les fonctions sont portées par les transitions et non pas par les états. En outre une et une seule transition est tirée par période d'exécution du composant, y compris des transitions vers l'état courant.

Le chapitre 6.1 explique comment s'effectue la transcription d'un modèle comportemental des composants à une modélisation temporelle des mêmes composants sous la forme de machines à états périodiques.

5.2 Traces

Les machines à états périodiques peuvent se « déplier » sous la forme d'un graphe de transitions possibles. Afin de produire une analyse temporelle précise, il faut calculer les temps de réponse des composants à partir des comportements possibles de leur machines à états. Pour cela, il faut représenter les évolutions possibles de la machine à états périodique sous la forme d'un ensemble de traces.

Le chapitre 6.2 décrit le protocole d'extraction de traces à partir des machines à états périodiques.

5.3 Borne supérieure des traces

La validité de l'analyse temporelle réside dans le fait qu'elle ne doit pas être *optimiste*, c'est à dire qu'elle ne doit pas sous-estimer le temps de réponse des composants. Une analyse optimiste ne permet donc pas de garantir le respect des échéances de chaque composant. Pour cela, le calcul de temps de réponse ne doit être obtenu qu'à partir de traces représentatives du pire cas d'exécution possible de la machine à états des composants. Il faut donc obtenir une borne supérieure de l'ensemble de traces.

Le chapitre 6.2 présente la méthode utilisée afin de calculer la borne supérieure des traces pour chaque composant.

5.4 Pire temps de réponse

Le calcul du pire temps de réponse d'un composant repose sur un processus itératif, un calcul de point fixe. Celui-ci utilise la borne supérieure des traces du composant étudié ainsi que les bornes supérieures des traces des composants de priorité plus élevée. On dit qu'un composant est ordonnançable si son pire temps de réponse est inférieur à son échéance.

Le calcul du pire temps de réponse est réitéré pour chaque composant de l'architecture afin de déterminer l'ordonnançabilité de celle-ci.

Le chapitre 6.3 décrit la méthodologie proposée afin d'exploiter les bornes supérieures des traces de chaque composant pour calculer les pire temps de réponse et décider de l'ordonnançabilité du système.

Chapitre 6

Conception d'une méthode d'analyse utilisant le modèle de tâches à base de machines à états

Sommaire

Résumé . *Afin d'analyser précisément le comportement temporel des composants constituant une architecture logicielle et de vérifier leur ordonnançabilité, nous procédons en trois étapes :*

1. *Chaque composant est modélisé sous la forme d'une* machine à états périodique. *Une machine à états périodique reprend les états et les transitions de la machine à états d'origine mais évalue et tire une transition à chaque période d'exécution du composant.*
2. *Les machines à états périodiques permettent une estimation précise du pire*

temps d'exécution des composants. Chaque machine à états périodique peut s'exécuter selon différents chemins de transitions et chaque chemin est représenté par une fonction de requête. *Une fonction de requête correspond au temps processeur cumulé passé dans la machine à états. Le pire temps d'exécution d'un composant est la borne supérieure des fonctions de requête provenant de la machine à états du composant.*

3. *L'évaluation du pire temps de réponse des composants se base sur des techniques éprouvées [*Liu, James *et* Layland *1973*; Spuri *1996*]. Ces techniques sont adaptées afin d'exploiter la précision qu'apporte les machines à états périodiques. Le pire temps de réponse de chaque composant est alors comparé à leur échéance pour décider de l'ordonnançabilité de l'architecture.*

Afin de garantir le déterminisme temporel lors de l'exécution d'un ensemble de tâches, il est nécessaire d'effectuer une analyse d'ordonnançabilité de l'architecture logicielle. L'objectif des analyses d'ordonnançabilité est de vérifier que chaque tâche respecte ses contraintes temporelles.

Ce chapitre est séparé en trois parties. La section 6.1 décrit comment la modélisation comportementale des composants est utilisée pour en faire un modèle temporel. Ensuite la section 6.2 présente l'utilisation du modèle temporel des tâches pour en extraire une estimation du pire temps d'exécution. Puis la section 6.3 décrit l'intégration de l'estimation du pire temps d'exécution dans des algorithmes de pire temps de réponse afin de vérifier l'ordonnançabilité de l'architecture. Enfin la section 6.4 présente une estimation des performances de cette méthode de calcul de pire temps de réponse en termes de complexité calculatoire et de précision.

6.1 Définition de *machine à états périodique*

L'objectif de cette section est de présenter la modélisation des tâches sous un angle différent. La modélisation proposée dans la partie I permet d'exprimer le comportement fonctionnel des tâches mais pas leur comportement temporel. Le modèle de tâches et de machines à états est donc modifié pour prendre en compte les informations temporelles.

Les tâches sont exécutées selon les spécifications du déploiement de l'architecture logicielle, à savoir un ensemble de tâches concurrentes périodiques. Ces tâches contiennent des machines à états représentant leur comportement fonctionnel, cependant une machine à états ne fournit pas d'information temporelle. Afin d'analyser finement le comportement temporel des tâches, nous modélisons les machines à états dites fonctionnelles sous la forme de *machines à états périodiques*. Une machine à états périodique possède le même comportement temporel que la tâche qu'elle modélise, en particulier sa période. Par

conséquent elle doit être capable de tirer une transition à chaque période, sans modifier le comportement fonctionnel de la machine à états d'origine. Pour cela, chaque état de la machine à états périodique doit posséder une transition vers lui-même.

Définition 6.1. *Une machine à états périodique est définie comme un ensemble d'états S identiques à ceux de la machine à états d'origine (voir l'équation (4.2)) et d'un ensemble de transitions Σ constitué des transitions de la machine à états d'origine auxquels s'ajoutent des transitions d'un état vers lui-même :*

$$\Sigma = E \cup \{s \to s \,|\, s \in S\} \tag{6.1}$$

Les composants définis sans machine à états sont modélisés comme des machines à états ne possédant qu'un seul état et une seule transition. La méthode `run` *de cet unique état est définie comme étant la fonction* `update` *du composant d'origine. Les autres méthodes sont laissées vides.*

L'ensemble de transitions Σ contient toutes les transitions de la machine à états E auxquelles s'ajoutent toutes les boucles sur les états de S.

Propriété 6.1. *Les machines à états périodiques tirent une transition à chaque période d'exécution.*

Afin d'éviter à l'analyse d'ordonnançabilité présentée dans le chapitre 6.3 de se bloquer dans des états puits, tous les états de la machine à états périodique doivent être accessibles à partir de n'importe quel autre état. Cette hypothèse est en général une bonne pratique à appliquer lors du développement de tâches : elle permet de remettre la tâche à son état initial ou à un état par défaut lors de l'exécution en cas de comportement erroné. En pratique cela revient à augmenter la *connexité* de la machine à états en ajoutant à chaque état une transition vers un état par défaut. Une machine à états est dite *connexe* lorsque chaque état peut être atteint à partir de n'importe quel état. De plus, notre modèle de composant indique que les composants peuvent être arrêtés et redémarrés selon les besoins de l'architecture, donc les machines à états modélisées au sein de ces composants possèdent implicitement des transitions vers l'état initial.

Propriété 6.2. *Les machines à états périodiques sont connexes : chaque état peut être atteint à partir de n'importe quel état par l'intermédiaire d'une séquence de transitions.*

$$\forall s_i, s_j \in S, \exists \sigma_1 \dots \sigma_n \in \Sigma \mid s_i \xrightarrow{\sigma_1} \dots \xrightarrow{\sigma_n} s_j \tag{6.2}$$

De plus nous avons choisi d'exprimer le temps de calcul des différentes méthodes (*entry*, *run*, *handle* et *exit*) de la machine à états dans les transitions de la machine à

états périodique. Par conséquent tout le temps de calcul de la machine à états périodique est passé dans les transitions, les états ne représentant que la structure de la machine à états d'origine.

Définition 6.2. *Une machine à états périodique possède sur chaque transition une fonction de délai exploitant les temps de calculs des quatre méthodes* entry, run, handle *et* exit *:*

$$\forall \sigma \in \Sigma, s_i \xrightarrow{\sigma} s_j,\ \delta(\sigma) = \begin{cases} s_i \neq s_j : & \mathcal{C}(run_i) + \mathcal{C}(exit_i) + \mathcal{C}(entry_j) \\ s_i = s_j : & \mathcal{C}(run_i) + \mathcal{C}(handle_i) \end{cases} \tag{6.3}$$

où $\mathcal{C}(m_i)$ correspond au pire temps d'exécution de la méthode m de la tâche τ_i.

La fonction de délai δ associe à chaque transition de la machine à états périodique le pire temps d'exécution des méthodes impliquées dans ces transitions. (voir la Figure 4.1 pour une description du protocole de transition).

Exemple 6.1. *Afin de faciliter les illustrations dans les sections suivantes, prenons une machine à états périodique contenant deux états et quatre transitions (voir la figure 6.1). Dans cette machine à états les deux états s_1 et s_2 sont connectés par les transitions σ_3 et σ_4. Les transitions σ_1 et σ_2 bouclent respectivement sur les états s_1 et s_2. Les durées des transitions sont arbitraires.*

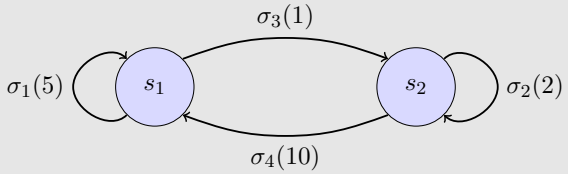


FIGURE 6.1 – Une machine à états périodique simplifiée

6.2 Développement d'une méthode de calcul de pire temps d'exécution par instance de machine à états périodique

Afin de réduire le pessimisme de l'estimation du pire temps d'exécution des tâches, nous allons présenter une méthode de calcul du temps d'exécution exploitant la structure des machines à états périodiques. Les machines à états périodiques permettent de modéliser finement le comportement des tâches en fonction du temps.

6.2.1 Traces

Une machine à états peut s'exécuter selon différents chemins dans le graphe représenté par les transitions et les états. Afin de pouvoir estimer précisément le temps processeur passé dans une machine à états périodique au cours du temps, nous définissons le concept de *trace*.

Définition 6.3. *Une* trace $\mathcal{T}$ *consiste en une séquence ordonnée de transitions de la machine à états périodique :*

$$\mathcal{T} = \langle \sigma_1, \ldots, \sigma_N \rangle \tag{6.4}$$

6.2.1.1 Opérations élémentaires et notations

Afin d'utiliser les traces dans les analyses d'ordonnancement, nous avons besoin de connaître le nombre de transitions impliquées dans une trace.

Définition 6.4. *La longueur* $|\mathcal{T}|$ *d'une trace* $\mathcal{T}$ *est donnée par le nombre de transitions* N *contenues dans celle-ci :*

$$|\mathcal{T}| = |\langle \sigma_1, \ldots, \sigma_N \rangle| = N \tag{6.5}$$

Les séquences de traces étant ordonnées, nous pouvons accéder à n'importe quel élément de la trace.

Définition 6.5. *La* i*-ème transition d'une trace est accessible grâce à l'opérateur* $\mathcal{T}[i]$ *:*

$$\mathcal{T}[i] = \langle \sigma_1, \ldots, \sigma_i, \ldots, \sigma_N \rangle [i] = \sigma_i \tag{6.6}$$

pour tout $i \in [\![1..N]\!]$*. Si* $i \notin [\![1..N]\!]$ *alors* $\mathcal{T}[i] = \emptyset$ *et* $\delta(\emptyset) = 0$*.*

De plus, le délai entre deux transitions consécutives d'une trace est nécessaire pour calculer le temps d'exécution des machines à états périodiques.

Définition 6.6. *Le délai entre les transitions* $i-1$ *et* i *est obtenu en utilisant la fonction de coût* $\Delta(\mathcal{T}, i)$ *:*

$$\Delta(\mathcal{T}, i) = \sum_{j=1}^{i} \delta(\mathcal{T}[j]) - \sum_{j=1}^{i-1} \delta(\mathcal{T}[j]) \tag{6.7}$$

Les transitions contenues dans une trace arbitraire $\mathcal{T}$ ne représentant pas nécessairement une exécution de la machine à états périodique, le délai entre deux transitions consécutives

$\mathcal{T}[i-1]$ et $\mathcal{T}[i]$ ne correspond pas obligatoirement au délai porté par la transition σ_i. L'équation 6.7 ne peut donc pas être simplifiée en $\Delta(\mathcal{T}, i) = \delta(\mathcal{T}[i])$.

Une trace est construite comme la concaténation de transitions. Nous pouvons donc ajouter des transitions à une trace.

Définition 6.7. *L'ajout d'une transition σ_M à la fin de la trace $\mathcal{T}$ est effectué avec l'opérateur $\langle \mathcal{T}, \sigma_M \rangle$:*

$$\langle \mathcal{T}, \sigma_M \rangle = \langle \sigma_1, \dots \sigma_N, \sigma_M \rangle \tag{6.8}$$

Dans le but de simplifier les équations suivantes, d'autres opérateurs de concaténation peuvent être définis comme $\langle \sigma_i, \mathcal{T} \rangle$ pour ajouter la transition σ_i au début de la trace $\mathcal{T}$ ou $\langle \sigma_i, \mathcal{T}, \sigma_M \rangle$ pour ajouter les transitions σ_i et σ_M respectivement au début et à la fin de la trace $\mathcal{T}$.

$$\begin{aligned} \langle \sigma_i, \mathcal{T} \rangle &= \langle \sigma_i, \sigma_1, \dots \sigma_N \rangle \\ \langle \sigma_i, \mathcal{T}, \sigma_M \rangle &= \langle \sigma_i, \sigma_1, \dots \sigma_N, \sigma_M \rangle \end{aligned} \tag{6.9}$$

Les transitions permettent le passage d'un état de départ à un état d'arrivée de la machine à états périodique.

Définition 6.8. *Les états de départ et d'arrivée de la transition σ sont accessibles par les opérateurs $from(\sigma)$ et $to(\sigma)$:*

$$\left. \begin{aligned} from(\sigma) \in S \\ to(\sigma) \in S \end{aligned} \right| from(\sigma) \xrightarrow{\sigma} to(\sigma) \tag{6.10}$$

Par extension, les traces sont des séquences de transitions partant d'un état de départ et se terminant dans un état d'arrivée.

Définition 6.9. *Les opérateurs $from(\mathcal{T})$ et $to(\mathcal{T})$ représentent respectivement l'état de départ et l'état d'arrivée d'une trace :*

$$\begin{aligned} from(\mathcal{T}) &= from(\mathcal{T}[1]) \\ to(\mathcal{T}) &= to(\mathcal{T}[|\mathcal{T}|]) \end{aligned} \tag{6.11}$$

Traces faisables Une trace est *faisable* si chaque paire consécutive de transitions est faisable.

Définition 6.10. *Une trace faisable ϕ est une trace dont l'état d'arrivée de chaque*

transition est l'état de départ de la suivante :

$$\forall i \in [\![1..|\phi - 1|]\!], to(\phi[i]) = from(\phi[i+1]) \tag{6.12}$$

Propriété 6.3. *Lorsque les traces sont faisables, la fonction de coût $\Delta(\phi, i)$ associée à ces traces s'écrit : $\Delta(\phi, i) = \delta(\phi[i])$*

Démonstration. Lorsque la trace ϕ ne contient qu'une transition, la preuve est triviale :

$$\begin{aligned} \phi = \langle\sigma\rangle \qquad \Delta(\phi, |\phi|) &= \delta(\sigma) - \delta(\emptyset) \\ &= \delta(\sigma) \\ &= \delta(\phi, |\phi|) \end{aligned} \tag{6.13}$$

La fonction de délai $\Delta(\phi, i) \mid i \in [\![1..|\phi|]\!]$ de la trace ϕ étant calculée à partir des transitions σ_1 à σ_i et si $\Delta(\phi', |\phi'|) = \delta(\phi', |\phi'|)$ tel que $|\phi'| = i$ alors $\Delta(\phi, i) = \delta(\phi, i)$.

$$\begin{aligned} \phi = \langle\mathcal{T}, \sigma\rangle \qquad \Delta(\phi, |\phi|) &= \sum_{i=1}^{|\phi|} \delta(\phi[i]) - \sum_{i=1}^{|\mathcal{T}|} \delta(\phi[i]) \\ &= \sum_{i=1}^{|\mathcal{T}|} \delta(\phi[i]) + \delta(\sigma) - \sum_{i=1}^{|\mathcal{T}|} \delta(\phi[i]) \\ &= \delta(\sigma) \\ &= \delta(\phi, |\phi|) \end{aligned} \tag{6.14}$$

La fonction de coût $\Delta(\phi, i)$ peut donc s'écrire $\Delta(\phi, i) = \delta(\phi, i)$ pour toute trace faisable ϕ. □

Fonction de requête La *fonction de requête* (rbf) d'une trace [Baruah, Rosier et Howell 1990] correspond à la requête processeur maximale d'une tâche sur un intervalle de temps. Les tâches s'exécutant de manière périodique, une fonction de requête est donc une fonction constante par morceaux. La hauteur des pas dépend des temps d'exécution des transitions et donc des pires temps d'exécution des méthodes des tâches.

Définition 6.11. *La fonction de requête $rbf(\mathcal{T}, t)$ d'une trace $\mathcal{T}$ au temps t est une*

fonction constante par morceaux :

$$rbf\left(\mathcal{T},t\right)=\sum_{i=0}^{\left\lfloor\frac{t}{T}\right\rfloor}\Delta\left(\mathcal{T},i+1\right) \tag{6.15}$$

Exemple 6.2. *Soit la trace $\mathcal{T}=\langle\sigma_1,\sigma_3,\sigma_2\rangle$ de la machine à états périodique présentée dans l'exemple 6.1. Le calcul de $rbf\left(\mathcal{T},t\right)$ se déroule comme suit :*

$$\begin{aligned}
t\in[0;T[&\implies rbf\left(\mathcal{T},t\right)=\Delta\left(\mathcal{T},1\right) &&=5\\
t\in[T;2\times T[&\implies rbf\left(\mathcal{T},t\right)=\Delta\left(\mathcal{T},1\right)+\Delta\left(\mathcal{T},2\right) &&=6\\
t\in[2\times T;3\times T[&\implies rbf\left(\mathcal{T},t\right)=\Delta\left(\mathcal{T},1\right)+\Delta\left(\mathcal{T},2\right)+\Delta\left(\mathcal{T},3\right) &&=8
\end{aligned} \tag{6.16}$$

La fonction de requête $rbf\left(\mathcal{T},t\right)$ peut donc être représentée par la figure 6.2.

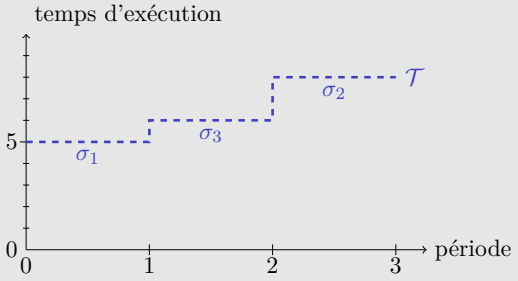


FIGURE 6.2 – Fonction de requête d'une des traces de la machine à états périodique

Relation d'ordre partiel entre les traces Afin d'utiliser les traces dans le calcul du pire temps d'exécution des tâches, nous avons besoin de définir dans quelles conditions une trace est « pire » qu'une autre, c'est à dire lorsqu'une trace est supérieure à une autre.

Définition 6.12. *Une trace $\mathcal{T}$ est supérieure à une autre trace $\mathcal{T}'$ si, pour tout instant t, la fonction de requête de $\mathcal{T}$ est supérieure à la fonction de requête de $\mathcal{T}'$. La relation de comparaison entre traces $\leq$ est donc définie par :*

$$\mathcal{T}\leq\mathcal{T}'\Leftrightarrow\forall t,rbf\left(\mathcal{T},t\right)\leq rbf\left(\mathcal{T}',t\right) \tag{6.17}$$

Lemme 6.1. *La relation $\leq$ définit un ordre partiel entre traces.*

Démonstration. La relation $\leq$ est réflexive : une trace est supérieure ou égale à elle-même,

comme le montre l'équation 6.18.

$$\forall \mathcal{T}, \forall t, rbf\left(\mathcal{T}, t\right) \leq rbf\left(\mathcal{T}, t\right) \tag{6.18}$$

De plus, la relation $\leq$ indique que lorsqu'une trace $\mathcal{T}$ est supérieure à une autre trace $\mathcal{T}'$ et que $\mathcal{T}'$ est supérieure à une troisième trace $\mathcal{T}''$, alors la trace $\mathcal{T}$ est supérieure à la trace $\mathcal{T}''$:

$$\begin{aligned} \forall \left(\mathcal{T}, \mathcal{T}', \mathcal{T}''\right), \forall t, rbf\left(\mathcal{T}'', t\right) \leq rbf\left(\mathcal{T}', t\right) \wedge rbf\left(\mathcal{T}', t\right) \leq rbf\left(\mathcal{T}, t\right) \\ \implies rbf\left(\mathcal{T}'', t\right) \leq rbf\left(\mathcal{T}, t\right) \end{aligned} \tag{6.19}$$

□

Exemple 6.3. *Soit $\mathcal{T} = \langle \sigma_1, \sigma_3, \sigma_2 \rangle$ et $\mathcal{T}' = \langle \sigma_1, \sigma_1, \sigma_3 \rangle$ deux traces de la machine à états périodique de l'exemple 6.1. La fonction de requête de chaque trace est calculée puis comparée afin de déterminer la trace supérieure.*

$$\begin{array}{lllllll} t \in [0; T[& \implies & rbf\left(\mathcal{T}, t\right) = 5 & ; & rbf\left(\mathcal{T}', t\right) & = 5 \\ t \in [T; 2 \times T[& \implies & rbf\left(\mathcal{T}, t\right) = 6 & ; & rbf\left(\mathcal{T}', t\right) & = 10 \\ t \in [2 \times T; 3 \times T[& \implies & rbf\left(\mathcal{T}, t\right) = 8 & ; & rbf\left(\mathcal{T}', t\right) & = 11 \end{array} \tag{6.20}$$

Dans ce cas, la trace $\mathcal{T}'$ est supérieure à la trace $\mathcal{T}$, comme le montre la figure 6.3.

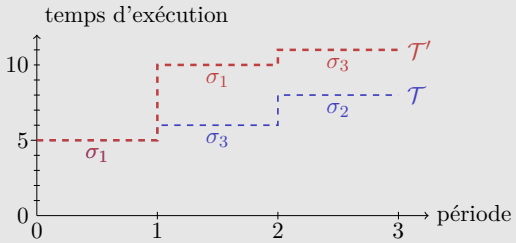


FIGURE 6.3 – Comparaison de fonctions de requête de deux traces de la machine à états périodique

Opérateur *next* Nous avons besoin de définir l'opérateur *next* représentant l'ensemble de traces suivant une trace spécifique afin de pouvoir construire récursivement les traces.

Définition 6.13. *L'opérateur next est l'ensemble de toutes les traces faisables commençant par la trace $\mathcal{T}$ et contenant une transition de plus :*

$$next\left(\mathcal{T}\right) = \{\langle \mathcal{T}, \sigma \rangle \mid \sigma \in \Sigma \wedge to\left(\mathcal{T}\right) = from\left(\sigma\right)\} \tag{6.21}$$

6.2.1.2 Borne supérieure des traces

Afin de calculer le pire temps d'exécution d'une tâche, nous avons besoin de calculer une borne supérieure de la fonction de demande des traces faisables de cette tâche. Pour cela, nous devons calculer les traces faisables de la tâche et en extraire une trace maximisant les fonctions de requêtes des traces faisables.

Construction d'un ensemble de traces $\mathcal{U}$ Le calcul d'un ensemble de traces est un processus itératif. La première itération de ce calcul se situe lors de l'instant critique, cependant cet instant est inconnu : les tâches ne sont pas synchronisées 4.4 lors de leur réveil initial. L'état dans lequel se trouve la machine à états d'une tâche est connu lors de son réveil mais ne peut pas être déterminé lors de l'instant critique. À l'état initial, l'ensemble de traces contient donc toutes les traces contenant une seule transition. Ensuite les transitions faisables sont ajoutées. Toutes les traces de l'ensemble de traces ont donc par construction le même nombre de transitions à chaque itération.

Définition 6.14. *L'ensemble de traces $\mathcal{U}^n$ est construit de manière itérative pour contenir toutes les traces de longueur n :*

$$\begin{aligned} \mathcal{U}^1 &= \{\langle\sigma\rangle \mid \sigma \in \Sigma\} \\ \mathcal{U}^{n+1} &= \bigcup_{\mathcal{T}\in\mathcal{U}^n} next\left(\mathcal{T}\right) \end{aligned} \tag{6.22}$$

Toutes les traces étant construites avec l'opérateur *next*, toutes les traces de l'ensemble sont faisables.

Exemple 6.4. *Pour la machine à états périodique de l'exemple 6.1, la première itération de l'ensemble de traces $\mathcal{U}^1$ contient toutes les transitions de la machine à états périodique :*

$$\mathcal{U}^1 = \{\langle\sigma_1\rangle, \langle\sigma_2\rangle, \langle\sigma_3\rangle, \langle\sigma_4\rangle\} \tag{6.23}$$

Pour l'itération suivante, l'ensemble de traces $\mathcal{U}^2$ possède toutes les paires de transitions commençant par celles contenues dans l'ensemble de traces $\mathcal{U}^1$. Chacun des quatre états possédant deux transitions, la taille de l'ensemble $\mathcal{U}^2$ est de $4^2 = 8$ éléments :

$$\mathcal{U}^2 = \{\langle\sigma_1,\sigma_1\rangle, \langle\sigma_1,\sigma_3\rangle, \langle\sigma_2,\sigma_2\rangle, \langle\sigma_2,\sigma_4\rangle, \langle\sigma_3,\sigma_2\rangle, \langle\sigma_3,\sigma_4\rangle, \langle\sigma_4,\sigma_1\rangle, \langle\sigma_4,\sigma_3\rangle\} \tag{6.24}$$

Lemme 6.2. *La taille de l'ensemble de traces $\mathcal{U}$ augmente exponentiellement avec la connexité de la machine à états périodique et donc avec le nombre d'états.*

Démonstration. Soit une machine à états périodique possédant N états et M transitions tel que le graphe représentant cette machine à états soit complet, à savoir $M = \frac{N(N-1)}{2}$. Dans ces conditions, l'ensemble de traces $\mathcal{U}^1$ contient M traces. L'opérateur *next* est utilisé sur cet ensemble afin de construire l'ensemble $\mathcal{U}^2$ contenant $M \times M$ traces. Par extension, la taille de l'ensemble $\mathcal{U}^n$ possède M^n traces. □

En pratique, l'équation 6.22 est difficilement utilisable pour traiter des machines à états périodiques de plus de quelques états à cause de la taille de l'ensemble $\mathcal{U}$. Afin de calculer la borne supérieure des traces sur des machines à états périodiques contenant un grand nombre d'états, nous proposons une version optimisée de l'équation 6.22.

Optimisation du calcul de l'ensemble de traces Le calcul du pire temps de réponse des tâches est basé sur le calcul d'une borne supérieure de toutes les traces faisables d'une tâche. Toute les traces ne sont donc pas intéressantes et nous pouvons donc raisonner sur les traces faisables maximales selon la relation 6.17. L'équation 6.22 indique que deux traces arrivant au même état vont être étendues de la même manière avec l'opérateur *next*. Nous construisons par conséquent un ensemble de traces réduit $\mathcal{V}$ tel que $\mathcal{V} \subset \mathcal{U}$ contenant seulement les traces faisables maximales.

Lemme 6.3. *Si deux traces se terminent dans le même état et une des deux est supérieure à l'autre dans le sens de la définition 6.12 alors seule celle possédant la plus grande valeur de fonction de requête peut être conservée dans l'ensemble de traces réduit.*

Démonstration. Soit deux traces $\mathcal{T} \in \mathcal{V}^n$ et $\mathcal{T}' \in \mathcal{V}^n \setminus \{\mathcal{T}\}$ se terminant dans le même état $s \in S \mid to(\mathcal{T}) = to(\mathcal{T}') = s$ définies telles que $\mathcal{T} \geq \mathcal{T}'$. Soit $\sigma \in \Sigma$ la transition possédant la plus grande durée à partir de s telle que $\sigma = \text{argmax}_{\sigma_i \in \Sigma} \delta(\sigma_i) \wedge to(\sigma) = s$ [1]. Soit les traces $\mathcal{T}'' \in next(\mathcal{T})$ et $\mathcal{T}''' \in next(\mathcal{T}')$ telles que $\mathcal{T}'' = \langle \mathcal{T}, \sigma \rangle$ et $\mathcal{T}''' = \langle \mathcal{T}', \sigma \rangle$.

Nous avons donc :

$$\begin{aligned} rbf(\mathcal{T}'', (n+1) \times T) &= rbf(\mathcal{T}, n \times T) + \delta(\sigma) \\ rbf(\mathcal{T}''', (n+1) \times T) &= rbf(\mathcal{T}', n \times T) + \delta(\sigma) \end{aligned} \tag{6.25}$$

1. argmax $f(x)$ est une fonction fournissant la valeur de x telle que $f(x)$ est maximale

Puisque $rbf\left(\mathcal{T}, n \times T\right) \geq rbf\left(\mathcal{T}', n \times T\right)$ alors

$$\begin{aligned} rbf\left(\mathcal{T}, n \times T\right) + \delta\left(\sigma\right) &\geq rbf\left(\mathcal{T}', n \times T\right) + \delta\left(\sigma\right) \\ rbf\left(\mathcal{T}'', (n+1) \times T\right) &\geq rbf\left(\mathcal{T}''', (n+1) \times T\right) \end{aligned} \tag{6.26}$$

Comme le montre l'équation 6.26, deux traces se terminant dans le même état et possédant des valeurs de fonction de requête différentes sont étendues à partir de la trace possédant la plus grande valeur de fonction de requête. □

Il est donc possible d'optimiser l'équation du calcul de l'ensemble de traces en ne prenant pas en compte les traces amenant à des traces inférieures.

Définition 6.15. *L'ensemble de traces $\mathcal{V}^n$ est construit de manière itérative pour contenir les uniques traces maximales de longueur n amenant à chaque état de la machine à états :*

$$\begin{aligned} \mathcal{V}^1 &= \left\{\left\langle\sigma\right\rangle, \sigma \in \Sigma \mid \forall \sigma' \in \Sigma, \sigma \neq \sigma' \wedge \left(to\left(\sigma\right) = to\left(\sigma'\right) \implies \sigma \geq \sigma'\right)\right\} \\ \mathcal{V}^{n+1} &= \left\{\left\langle\mathcal{T}, \sigma\right\rangle, \mathcal{T} \in \mathcal{V}^n, \sigma \in \Sigma \mid \forall \mathcal{T}' \in \mathcal{V}^n, \forall \sigma' \in \Sigma, \mathcal{T}' \neq \mathcal{T}\right. \\ &\qquad \wedge \left\langle\mathcal{T}, \sigma\right\rangle \in next\left(\mathcal{T}\right) \wedge \left\langle\mathcal{T}', \sigma'\right\rangle \in next\left(\mathcal{T}'\right) \\ &\qquad \left. \wedge \ \left(to\left(\left\langle\mathcal{T}, \sigma\right\rangle\right) = to\left(\left\langle\mathcal{T}', \sigma'\right\rangle\right) \implies \left\langle\mathcal{T}, \sigma\right\rangle \geq \left\langle\mathcal{T}', \sigma'\right\rangle\right)\right\} \end{aligned} \tag{6.27}$$

Exemple 6.5. *Pour la machine à états périodique de l'exemple 6.1, la première itération de l'ensemble de traces $\mathcal{V}^1$ contient les transitions de la machine à états périodique telles que pour chaque état, seules les transitions maximales amenant à ces états sont ajoutés. Pour l'état s_1, les transitions amenant à cet état sont σ_1 et σ_4. Les durées de ces transitions valent respectivement 5 et 10 unités de temps donc seule la transition σ_4 est ajoutée à l'ensemble de traces $\mathcal{V}^1$. Ce processus est répété pour l'état s_2 :*

$$\mathcal{V}^1 = \left\{\left\langle\sigma_2\right\rangle, \left\langle\sigma_4\right\rangle\right\} \tag{6.28}$$

Les itérations suivantes sont construites en ajoutant aux traces les transitions les plus coûteuses en terme de temps cumulé. La figure 6.4 montre ce processus : les transitions provenant de s_1 et s_2 sont testées dans l'ensemble de traces $\mathcal{V}^1$. Les états d'arrivée de ces transitions sont d'autres instances de s_1 et s_2. La transition σ_1 possède une durée moins importante que σ_4 donc elle n'est pas ajoutée à l'ensemble de traces. Ce processus est répété pour σ_2 et σ_3.

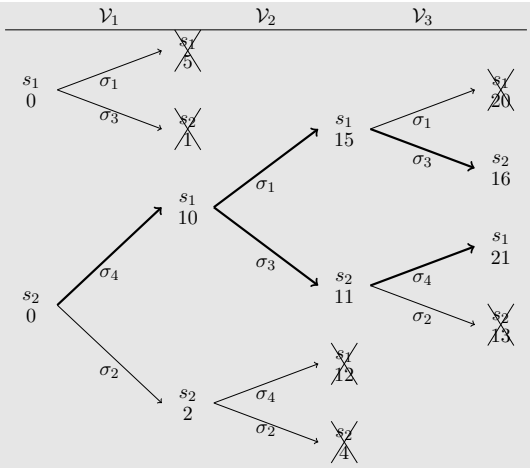


FIGURE 6.4 – Valeurs de l'ensemble de traces $\mathcal{V}^n$ sur trois itérations

Pour l'itération suivante, l'ensemble de traces $\mathcal{V}^2$ possède seulement les paires de transitions commençant par celles contenues dans l'ensemble de traces $\mathcal{V}^1$:

$$\mathcal{V}^2 = \{\langle \sigma_4, \sigma_1 \rangle, \langle \sigma_4, \sigma_3 \rangle\} \tag{6.29}$$

La troisième itération, $\mathcal{V}^3$, comporte elle aussi seulement deux trace mais de trois transitions :

$$\mathcal{V}^3 = \{\langle \sigma_4, \sigma_1, \sigma_3 \rangle, \langle \sigma_4, \sigma_3, \sigma_4 \rangle\} \tag{6.30}$$

Borne supérieure des traces Un ensemble de traces représente toutes les exécutions possibles de la machine à états d'une tâche. Nous cherchons une borne supérieure $\mathcal{T}^+$ à partir d'un ensemble de traces qui vérifie l'équation 6.31.

$$\mathcal{T}^+ : \forall n, \forall \mathcal{T} \in \mathcal{V}^n \mid \mathcal{T}^+ \geq \mathcal{T} \tag{6.31}$$

Cette borne supérieure $\mathcal{T}^+$ est représentée sous la forme d'une trace pour des raisons pratiques d'implémentation et permet d'exprimer le pire temps d'exécution par période de la machine à état. Nous pouvons noter que la borne supérieure des traces n'est pas nécessairement une trace faisable.

Définition 6.16. *La trace $\mathcal{T}^+$ est construite de manière itérative afin de maximiser*

l'ensemble de traces $\mathcal{U}$ *:*

$$\begin{aligned} \mathcal{T}_1^+ &= \langle \sigma \rangle \mid \sigma = \underset{\sigma_i \in \Sigma}{\operatorname{argmax}}\, \delta(\sigma_i) \\ \mathcal{T}_{n+1}^+ &= \left\langle \mathcal{T}_n^+, \mathcal{T}[n+1] \right\rangle \mid \mathcal{T} = \underset{\mathcal{T}' \in \mathcal{V}^{n+1}}{\operatorname{argmax}}\, rbf(\mathcal{T}', (n+1) \times T) \end{aligned} \tag{6.32}$$

La fonction de délai $\Delta(\mathcal{T}^+, i)$ associée à borne supérieure $\mathcal{T}^+$ vaut donc :

$$\Delta\left(\mathcal{T}^+, i\right) = \max_{\mathcal{T} \in \mathcal{V}^i} rbf(\mathcal{T}, i \times T) - rbf(\mathcal{T}, (i-1) \times T) \tag{6.33}$$

Lemme 6.4. *La trace* $\mathcal{T}^+$ *construite par l'équation 6.32 est une borne supérieure de l'ensemble de traces réduit* $\mathcal{V}$*.*

Démonstration. Pour la première itération de l'équation 6.32, la trace $\mathcal{T}_1^+$ est construite comme une trace d'une seule transition maximisant le délai des transitions de la machine à états périodique.

Pour les itérations suivantes, si $\mathcal{T}_n^+$ est une borne supérieure de l'ensemble de traces réduit $\mathcal{V}^n$ tel que $\forall \mathcal{T}_n \in \mathcal{V}^n, \mathcal{T}_n^+ \geq \mathcal{T}_n$ alors :

$$\begin{aligned} rbf(\mathcal{T}_{n+1}, (n+1) \times T) &= rbf(\langle \mathcal{T}_n, \mathcal{T}_{n+1}[n+1] \rangle, (n+1) \times T) \\ &= rbf(\mathcal{T}_n, n \times T) + \Delta(\mathcal{T}_{n+1}, n+1) \\ &\leq rbf\left(\mathcal{T}_n^+, n \times T\right) + \Delta(\mathcal{T}_{n+1}, n+1) \end{aligned} \tag{6.34}$$

sachant que par construction, $\Delta(\mathcal{T}_{n+1}, n+1)$ ne peut pas être supérieur à $\Delta\left(\mathcal{T}_{n+1}^+, n+1\right)$ alors

$$rbf\left(\mathcal{T}_{n+1}^+, (n+1) \times T\right) \geq rbf(\mathcal{T}_{n+1}, (n+1) \times T) \tag{6.35}$$

donc $\mathcal{T}_{n+1}^+$ est la borne supérieure de $\mathcal{V}^{n+1}$. □

Exemple 6.6. *La figure 6.5 illustre la construction de la borne supérieure des traces. Les deux traces de l'ensemble* $\mathcal{V}^3$ *s'entrelacent mais seule la trace les maximisant est conservée.*

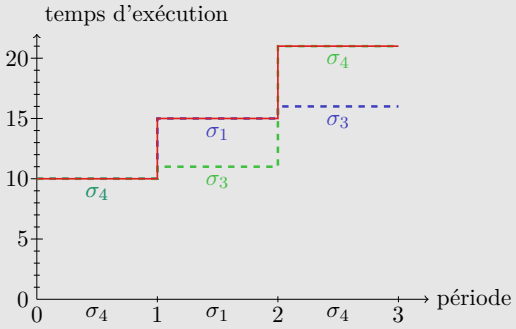


FIGURE 6.5 – Représentation de $\mathcal{V}^n$ sur trois itérations en utilisant l'équation 6.27

La borne supérieure des traces est représentée par la trace $\mathcal{T}^+ = \langle \sigma_4, \sigma_1, \sigma_4 \rangle$ *qui n'est pas une trace faisable par la machine à états périodique, comme le montre la figure 6.6.*

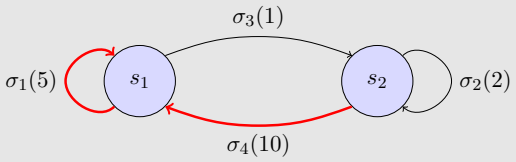


FIGURE 6.6 – La borne supérieure des traces n'est pas une trace faisable

L'algorithme 1 calcule un ensemble de traces $\mathcal{V}^n$ de longueur n ainsi que la borne supérieure $\mathcal{T}^+$ d'une tâche en même temps. Cet algorithme est basé sur l'équation optimisée 6.27 et introduit une nouvelle optimisation d'implémentation. En effet, il est possible de ne conserver qu'une trace par itération en construisant la borne supérieure $\mathcal{T}^+$ à chaque pas de calcul. Par conséquent le nombre de transitions utilisées pour le calcul de $\mathcal{T}^+$ se limite à $n \times |S|$.

La première étape de l'algorithme consiste à initialiser la borne supérieure des traces $\mathcal{T}^+$ (ligne 1) et l'ensemble de traces $\mathcal{V}$ (ligne 2) à des ensembles vides. Le dictionnaire $X(s)$ liant une trace $\mathcal{T}$ à un état s est aussi initialisé (lignes 3 à 10) : la transition la plus coûteuse en temps amenant à chaque état est ajoutée au dictionnaire. Ensuite, pour chaque état de la machine à états, nous ajoutons à l'ensemble $\mathcal{V}$ la trace d'une transition par l'intermédiaire du dictionnaire X (ligne 12). La borne supérieure $\mathcal{T}^+$ est sélectionnée à partir de la trace de X la plus coûteuse (ligne 14). Les itérations suivantes sont calculées de manière similaire : pour chaque trace $\mathcal{T}$ de l'ensemble $\mathcal{V}$, le dictionnaire X est mis à jour avec les traces $\mathcal{T}'$ suivant $\mathcal{T}$ avec l'opérateur *next* (ligne 21). X contient alors toutes les traces maximales amenant à chaque état de la machine à états. L'ensemble $\mathcal{V}$ est alors reconstruit à partir du dictionnaire mis à jour (ligne 28). Enfin la borne supérieure $\mathcal{T}^+$

Algorithme 1 Algorithme de calcul de la borne supérieure des traces

1: $\mathcal{T}^+ \leftarrow \langle\rangle$
2: $\mathcal{V} \leftarrow \{\}$
3: **pour tout** $s \in S$ **faire**
4: $\quad X(s) \leftarrow \langle\rangle$
5: **fin pour**
6: **pour tout** $\sigma \in \Sigma$ **faire**
7: $\quad$ **si** $\langle\sigma\rangle > X(to(\sigma))$ **alors**
8: $\quad\quad X(to(\sigma)) \leftarrow \langle\sigma\rangle$
9: $\quad$ **fin si**
10: **fin pour**
11: **pour tout** $s \in S$ **faire**
12: $\quad \mathcal{V} \leftarrow \mathcal{V} \cup X(s)$
13: $\quad$ **si** $X(s) > \mathcal{T}^+$ **alors**
14: $\quad\quad \mathcal{T}^+ \leftarrow X(s)$
15: $\quad$ **fin si**
16: **fin pour**
17: **pour tout** $i \in [\![2..N]\!]$ **faire**
18: $\quad$ **pour tout** $\mathcal{T} \in \mathcal{V}$ **faire**
19: $\quad\quad$ **pour tout** $\mathcal{T}' \in next(\mathcal{T})$ **faire**
20: $\quad\quad\quad$ **si** $\mathcal{T}' > X(to(\mathcal{T}'))$ **alors**
21: $\quad\quad\quad\quad X(to(\mathcal{T}')) \leftarrow \mathcal{T}'$
22: $\quad\quad\quad$ **fin si**
23: $\quad\quad$ **fin pour**
24: $\quad$ **fin pour**
25: $\quad \mathcal{V} \leftarrow \{\}$
26: $\quad \mathcal{T} \leftarrow \mathcal{T}^+$
27: $\quad$ **pour tout** $s \in S$ **faire**
28: $\quad\quad \mathcal{V} \leftarrow \mathcal{V} \cup X(s)$
29: $\quad\quad$ **si** $X(s) > \mathcal{T}$ **alors**
30: $\quad\quad\quad \mathcal{T} \leftarrow X(s)$
31: $\quad\quad$ **fin si**
32: $\quad$ **fin pour**
33: $\quad \mathcal{T}^+ \leftarrow \langle \mathcal{T}^+, \mathcal{T}[i] \rangle$
34: $\quad \Delta(\mathcal{T}^+, i) \leftarrow rbf(\mathcal{T}, i \times T) - rbf(\mathcal{T}^+, (i-1) \times T)$
35: **fin pour**

est complété par la dernière transition de la trace possédant un coût maximal (ligne 33) et nous définissons la fonction de coût associée à $\mathcal{T}^+$ (ligne 34).

Arrêt du calcul de la borne supérieure des traces Les processus récursifs 6.22 et 6.27 ainsi que l'algorithme 1 permettent de calculer la borne supérieure de la fonction de requête des tâches pour N itérations. La borne sur ce nombre d'itérations est définie dans la section 6.3.1.

6.3 Adaptation des méthodes de calcul de pire temps de réponse pour utiliser les machines à états périodiques

Nous avons proposé une méthode permettant de calculer une approximation de la fonction de requête d'une tâche en calculant une borne supérieure de traces de longueur N. Afin de compléter l'analyse d'ordonnançabilité, nous utilisons la borne supérieure des traces de chaque tâche pour en calculer le pire temps de réponse.

Cette section est séparée en deux : dans un premier temps, nous déterminons la longueur maximale des traces nécessaire au calcul du pire temps de réponse ; enfin nous adaptons les techniques de calcul du pire temps de réponse pour prendre en compte les traces d'exécution.

6.3.1 Calcul de la longueur de trace nécessaire au calcul du pire temps de réponse sur une architecture complète

Le réveil des tâches est effectué par le système d'exploitation. Tous les systèmes d'exploitation ne permettant pas de définir ou de contrôler les dates de réveil des tâches, il est impossible de connaître la première date de réveil R_i des tâches. Nous considérons alors que la date de réveil de chaque tâche peut se situer entre sa date d'activation et sa période.

Hypothèse 6.1. *La date de réveil des tâches est inconnue et comprise entre leur activation et leur période :*

$$\forall i, R_i \in [0..T_i] \tag{6.36}$$

L'hypothèse 6.1 indique que les dates réveil des tâches sont inconnues et les machines à états tirent une transition à chaque période selon la propriété 6.1. De plus, la propriété 6.2 indique que les machines à états sont fortement connexes. Il est donc possible pour chaque machine à états de se trouver dans n'importe quel état dans le futur. Chaque tâche peut donc atteindre n'importe quel état et y rester. En outre, toutes les tâches peuvent se réveiller au même moment parce que nous ne connaissons pas la date de réveil exacte des tâches.

Pour chaque tâche, la borne supérieure des traces est plus grande que toutes les traces d'exécution. Il est donc possible de trouver un instant critique où toutes les tâches commencent leur exécution au même moment avec la première transition de leur borne supérieure de traces.

Nous pouvons donc affirmer que pour chaque tâche, la première échéance est la plus restrictive [Joseph et Pandya 1986]. Nous limitons la plage d'étude au maximum des échéances :

$$D = \max_i D_i, \tag{6.37}$$

Les bornes supérieures des traces, pour chaque tâche i, sont alors calculées jusqu'à $N = \left\lceil \frac{D}{T_i} \right\rceil$.

6.3.2 Adaptation des méthodes de calcul de pire temps de réponse pour utiliser les machines à états périodiques

Pour pouvoir calculer le pire temps de réponse des tâches, nous avons modifié le processus récursif usuel appliqué à *Rate Monotonic* ou à *Deadline Monotonic* [Audsley et al. 1993].

Le modèle d'exécution des tâches présenté dans la section 4 indique que les communications entre tâches sont non-bloquantes (voir l'hypothèse 4.2). L'exécution des tâches n'est donc pas retardée par l'attente de données provenant d'autres tâches. Par conséquent, la méthode de calcul de pire temps de réponse n'intègre pas cette notion de retard d'exécution.

Afin d'exploiter les fonctions de requête des tâches calculées à partir des bornes supérieures des traces dans l'analyse d'ordonnançabilité, nous avons adapté ce processus récursif : au lieu d'utiliser le pire temps d'exécution des tâches, nous exploitons la borne supérieure des traces de ces tâches. Puis pour chaque itération du processus récursif, nous utilisons le pas suivant de la borne supérieure des traces.

Il est alors possible de calculer le pire temps de réponse $\mathcal{R}_i$ de la $i^{ème}$ tâche en initialisant sa valeur à $rbf\left(\mathcal{T}^+, 0\right)$, c'est à dire à la valeur de la première itération de la fonction de requête. Ensuite, le pire temps de réponse est incrémenté récursivement avec les temps d'exécution des instances de tâche de plus haute priorité.

$$\begin{aligned} \mathcal{R}_i^0 &= rbf\left(\mathcal{T}_i^+, 0\right) \\ \mathcal{R}_i^{n+1} &= \sum_{j \leq hp(i)} rbf\left(\mathcal{T}_j^+, \mathcal{R}_i^n\right) + \mathcal{R}_i^0 \end{aligned} \tag{6.38}$$

avec $hp(i)$ les instances de tâches de plus haute priorité que la tâche i. Deux conditions

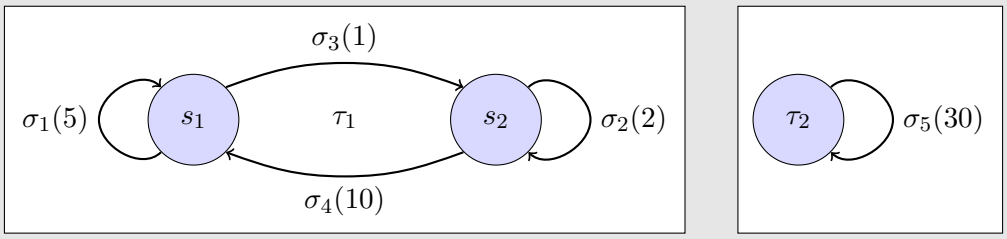
peuvent arrêter cette boucle récursive :

— lorsque deux itérations consécutives ont la même valeur $\mathcal{R}_i^n = \mathcal{R}_i^{n+1}$;

— lorsque $\mathcal{R}_i^{n+1}$ atteint l'échéance de la tâche i.

Dans le second cas, la tâche n'est pas ordonnançable. Si le pire temps de réponse de chaque tâche de l'architecture est inférieur à son échéance ($\forall i, \mathcal{R}_i \leq D_i$), le système est ordonnançable.

Exemple 6.7. *Dans cet exemple, nous reprenons la machine à états périodique présentée dans l'exemple 6.1 et l'associons à la tâche τ_1 avec les paramètres suivants : sa priorité est définie à 2, sa période et son échéance valent toutes deux 20 unités de temps. Ensuite, pour simplifier la compréhension, nous ajoutons la tâche τ_2 modélisée par une machine à états possédant un seul état. τ_2 possède une priorité de 1, inférieure à la priorité de τ_1 et sa période ainsi que son échéance sont définies à 60 unités de temps.*



$$P_1 = 1 \quad T_1 = 20 \quad D_1 = 20 \qquad P_2 = 2 \quad T_2 = 60 \quad D_2 = 60$$

FIGURE 6.7 – Architecture à deux tâches périodiques

La tâche τ_1 étant plus prioritaire que la tâche τ_2, son temps de réponse se résume à sa transition de durée la plus élevée : $\mathcal{R}_1^0 = \sigma_4 = 10$.

La tâche τ_2 possédant une priorité moins élevée que τ_1, elle subit les préemptions de τ_1 durant son exécution. Son temps de réponse est calculé dans l'équation suivante :

$$\begin{aligned}
\mathcal{R}_2^0 &= \delta(\sigma_5) &&= 30 \\
\mathcal{R}_2^1 &= rbf\left(\mathcal{T}_1^+, \mathcal{R}_2^0\right) + \delta(\sigma_5) = 15 + 30 &&= 45 \\
\mathcal{R}_2^2 &= rbf\left(\mathcal{T}_1^+, \mathcal{R}_2^1\right) + \delta(\sigma_5) = 21 + 30 &&= 51 \\
\mathcal{R}_2^3 &= rbf\left(\mathcal{T}_1^+, \mathcal{R}_2^2\right) + \delta(\sigma_5) = 21 + 30 &&= 51
\end{aligned} \tag{6.39}$$

Le calcul du pire temps de réponse pour ces deux tâches donne 10 unités de temps

pour la tâche τ_1 et 51 unités de temps pour la tâche τ_2.

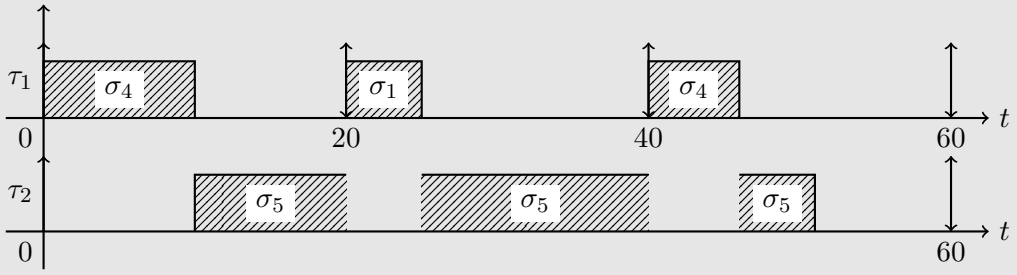


FIGURE 6.8 – Étude d'ordonnançabilité d'une architecture à deux tâches périodiques

6.4 Estimation des performances

Afin de déterminer les performances de cette méthode d'analyse utilisant les machines à états périodiques contenues dans les composants, nous devons estimer deux éléments : la complexité de l'analyse et le le gain de précision du calcul de pire temps de réponse.

6.4.1 Complexité de la méthode d'analyse utilisant le modèle de tâches à base de machines à états

Dans cette section nous estimons l'ajout de complexité de la méthode d'analyse proposée prenant en compte les machines à états des composants par rapport à une méthode *classique* considérant les composants comme des entités insécables. Le coût calculatoire est indiqué en opérations par étapes.

L'objectif est ici de déterminer la complexité de l'ensemble des opérations, à savoir l'extraction des machines à états périodiques, le calcul des traces puis de l'ensemble de traces, ensuite la détermination de la borne supérieure des traces et enfin le calcul du pire temps de réponse.

6.4.1.1 Extraction des machines à états périodiques

Pour chaque machine à états, le coût d'extraction de la machine à états périodique correspondante est linéaire : toutes deux contiennent les mêmes états et les mêmes transitions. La complexité de ce changement de modèle est donc $O(|S| + |\Sigma|)$.

6.4.1.2 Calcul des traces

Pour chaque machine à états périodique PSM_i du composant i, le nombre de traces dépend de quatre éléments :

1. de la période d'étude D du système ;
2. de la période T_i de la machine à états périodique ;
3. du nombre d'états $|S|$ de la machine à états périodique ;
4. du nombre de transitions $|\Sigma|$ de la machine à états périodique.

Pour chaque machine à états périodique PSM_i, le nombre de traces est borné par le nombre de chemins possibles de longueur $\left\lceil \frac{D}{T_i} \right\rceil$ dans le graphe que représente la machine à états périodique. Le coût de calcul d'une trace à partir d'une machine à états périodique dépend de sa connexité et de la longueur de la trace. Pour une machine à états périodique, sa connexité est majorée par son nombre d'états $|S|$.

La complexité du calcul de l'ensemble de traces complet $\mathcal{U}$ dépend de plus du nombre d'états $|S|$ de la machine à états périodique. La complexité du calcul de $\mathcal{U}$ vaut donc $O\left(|S|^{\left\lceil \frac{D}{T_i} \right\rceil}\right)$.

Cependant, le calcul de l'ensemble de traces réduit $\mathcal{V}$ ne conserve à chaque itération que l'ensemble d'états permettant de calculer la borne supérieure des traces. Pour cela, toutes les transitions de la machine à états périodique sont tirées. Par conséquent, le nombre de calculs effectués pour déterminer l'ensemble de traces réduit est borné par $O\left(|\Sigma| \times \left\lceil \frac{D}{T_i} \right\rceil\right)$.

6.4.1.3 Borne supérieure des traces

Le coût calculatoire de la borne supérieure des traces dépend du nombre de traces et de la longueur de celles-ci. Pour un composant i, la complexité du calcul de cette borne supérieure à partir d'un ensemble de traces est : $O\left(\left\lceil \frac{D}{T_i} \right\rceil\right)$.

6.4.1.4 Pire temps de réponse

Cette partie du calcul est identique à la méthode classique, elle n'ajoute donc pas de coût calculatoire supplémentaire.

La complexité ajoutée par l'utilisation de machines à états périodiques au sein du calcul de pire temps de réponse est résumé dans le tableau 6.1. La complexité de chaque étape est additionnée pour obtenir la complexité totale. Sachant que $\left\lceil \frac{D}{T_i} \right\rceil$ est toujours

Étape	Complexité
Extraction des machines à états périodiques	$\|S\| + \|\Sigma\|$
Calcul des traces	$\|\Sigma\| \times \left\lceil \frac{D}{T_i} \right\rceil$
Calcul de la borne supérieure des traces	$\left\lceil \frac{D}{T_i} \right\rceil$
Total	$\|S\| + \|\Sigma\| + \|\Sigma\| \times \left\lceil \frac{D}{T_i} \right\rceil + \left\lceil \frac{D}{T_i} \right\rceil$

TABLE 6.1 – Récapitulatif de la complexité des différentes étapes de calcul du pire temps de réponse

supérieur ou égal à 1 et qu'une machine à états périodique contient toujours plus d'une transition, la complexité totale peut être simplifiée en $O\left(|S| + |\Sigma| \times \left\lceil \frac{D}{T_i} \right\rceil\right)$.

Remarque 6.1. *D'un point de vue pratique, nous avons constaté que le temps de calcul de l'analyse temps-réel utilisant les machines à états périodiques est sensiblement identique par rapport à une méthode d'analyse utilisant des tâches canoniques :*

- *la dimension globale des éléments est faible. Le nombre d'états des machines à états périodiques est identique à celui des machines à états des composants fournis par le développeur et est souvent limité à quelques dizaines d'états. La connexité des machines à états est limitée par le nombre d'états. De plus, le nombre d'itérations nécessaires pour le calcul de l'analyse temps-réel dépend de l'hétérogénéité des périodes et des échéances des tâches impliquées dans l'architecture ;*
- *La propriété d'inclusion 6.2 permet de réduire le nombre de traces à prendre en compte lors de l'analyse.*

6.4.2 Estimation du gain de précision de la méthode utilisant les machines à états de tâches par rapport à celle utilisant les tâches monolithiques

La méthode présentée utilise les fonctions de requêtes des composants afin de calculer leur pire temps de réponse. Le gain de précision de cette méthode est donc quantifié par la fonction de requête, et plus précisément en comparant les fonctions de requêtes calculées à partir des machines à états périodiques et celles extraites de tâches canoniques.

6.4.2.1 Observations pratiques

La méthode présentée au chapitre 6 étudie l'ordonnançabilité d'un ensemble de composants ordonnancé par un ordonnanceur à priorité fixe en calculant leur pire temps

de réponse. Le calcul du pire temps de réponse de chaque composant τ_i dépend des composants possédant une priorité plus élevée et plus particulièrement des fonctions de requêtes des composants possédant une priorité plus élevée. La précision de notre méthode provient du calcul précis des fonctions de requêtes des composants en exploitant la structure de leur machine à états interne.

Dans un modèle de composant sans machine à états, la « pire » transition, c'est à dire la transition la plus temporellement coûteuse, est exécutée à chaque période. Dans notre modèle de composant, cette « pire » transition est exécutée à la première période, puis les autres transitions de la machine à états sont prises en compte. Dans ces conditions, plus la machine à états est complexe et comporte des temps d'exécution variés, plus le modèle temporel du composant est précis. La précision augmente d'autant plus que la borne supérieure des traces est calculée sur une longue période. Nous avons montré, sous nos hypothèses, que seule la première échéance importe pour l'analyse d'ordonnançabilité ; par conséquent la précision de notre méthode augmente aussi avec l'échéance du composant pour lequel nous calculons le pire temps de réponse.

Ces deux observations sont intéressantes en pratique puisqu'elles correspondent à des « bonnes pratiques » de systèmes temps-réel. Pour des systèmes à échéance sur requête, les priorités des tâches sont souvent dépendantes de leur période (avec un ordonnanceur *Rate Monotonic*). Dans le cas où les échéances des tâches sont inférieures à leurs périodes, les priorités des tâches sont souvent définies à partir de leur échéance (avec un ordonnanceur *Deadline Monotonic*). En appliquant ces « bonnes pratiques » les fonctions de requête des tâches sont calculées sur une durée plus longue et augmentent le gain de précision.

6.4.2.2 Indicateur

Le gain de précision de notre méthode est assez aisé à estimer par rapport à une méthode n'utilisant pas les machines à états des composants. La méthode d'analyse utilisant les tâches canoniques considère que chaque tâche utilise son pire temps d'exécution à chaque période. Ce temps correspond pour notre modèle à la première transition de la borne supérieure des traces, à savoir $rbf(\tau_i, 0)$. Pour chaque composant τ_i possédant une échéance D_i, le gain de précision vaut donc :

$$\sum_{t=0}^{D_i} \left(rbf(\tau_i, 0) - rbf(\tau_i, t) \right) \tag{6.40}$$

avec rbf_i^* la fonction de requête du composant considéré.

Exemple 6.8. *Ce processus est présenté dans la figure 6.9 : la transition la plus coûteuse en temps apparaît à la première itération lorsque la transition σ_4 est tirée,*

ce qui correspond aussi à la valeur de la fonction de requête de la tâche canonique, représentée par rbf^*.

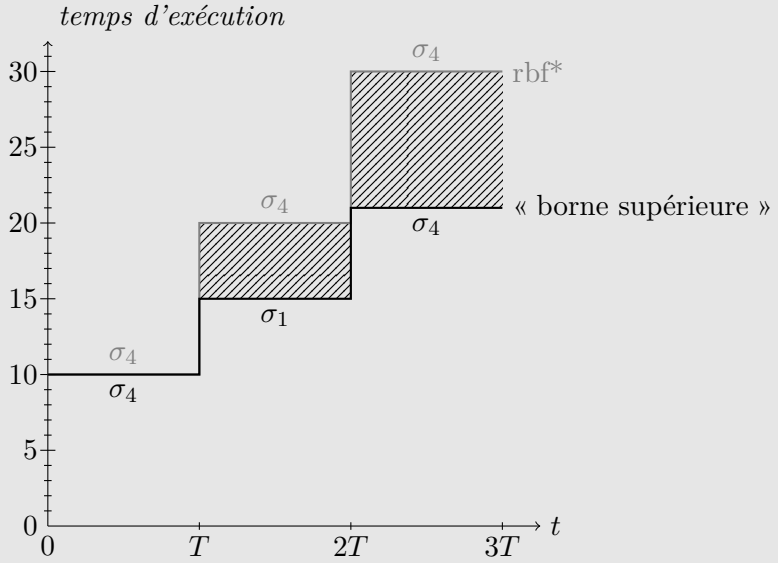


FIGURE 6.9 – Comparaison entre les valeurs de la borne supérieure des traces $\mathcal{U}^n$ et de la fonction de requête « classique » rbf^*

Cependant, lors de la deuxième itération, la machine à états tire la transition σ_1, même si elle n'est pas la transition la plus coûteuse, elle correspond tout de même à une borne supérieure du temps d'exécution de la tâche. Grâce à la précision apportée par la machine à états, nous avons gagné cinq unités de temps dès la deuxième itération par rapport à la fonction de requête « classique ». La troisième itération augmente encore plus cet écart fournissant ainsi une borne supérieure encore moins pessimiste.

6.5 Conclusion

Nous avons donc développé une méthode de calcul du temps d'exécution de tâches contenant des machines à états. Cette estimation se base sur le calcul de fonctions de requêtes à partir de traces d'exécution. Nous avons par la suite modifié les méthodes usuelles de calcul de pire temps de réponse pour exploiter la précision apportée par la modélisation des machines à états des tâches. De plus, le calcul des pires temps de réponse permet de décider de l'ordonnançabilité ou non de l'architecture logicielle. Enfin nous avons démontré le gain de performance en terme de précision de cette méthode par rapport aux techniques usuelles tout en maintenant le coût calculatoire à un niveau

raisonnable.

Afin de valider le fonctionnement et la capacité de cette méthode d'analyse en conditions réelles, nous avons développé et analysé une architecture robotique conçue pour des missions d'exploration de zone par un robot autonome. La partie suivante du manuscrit traite donc des étapes nécessaires à la génération et à l'exécution d'une telle architecture.

Troisième partie

Expérimentation

Chapitre 7

Développement d'un logiciel de génération de code exécutable utilisant le modèle de tâches à base de machines à états

Sommaire

Résumé . *Le langage de modélisation Mauve fournit des outils pour modéliser puis analyser des architectures logicielles à base de composants contenant des machines à états. Cependant, pour que les résultats d'analyse temps-réel restent valides par rapport à l'exécution réelle de ces architectures, il faut les transcrire en codes exécutables tout en respectant les spécifications fournies lors de la phase de modélisation. Pour cela nous utilisons un générateur de code exploitant les différentes couches des modèles afin de générer des composants compilables et exécutables par l'intermédiaire du middleware Orocos.*

Mauve fournit un ensemble d'outils de génération de code pour les composants, les architectures et les déploiements. Un générateur de code est un programme logiciel dont le rôle est de créer du code à partir de modèles existants. L'utilisation de générateurs de code permet de faciliter et d'accélérer le développement de programmes. Dans le cadre de cette thèse, nous avons conçu un générateur de code C++ destiné au middleware Orocos

utilisant les modèles de composants, d'architectures et de déploiements présentés dans le chapitre 3. La conception de ce type d'outils a trois avantages principaux :

— le code généré est conforme à la spécification du modèle ;
— le développement de composants, d'architectures et de déploiements est accéléré par la diminution de la quantité de code à écrire ;
— le générateur de code ajoute automatiquement un système de traces afin de pouvoir obtenir des informations sur l'exécution des composants.

La génération complète de code à partir de modèles de tâches jusqu'aux exécutables compilés se compose de six éléments : les fichiers contenant les modèles de tâches sont utilisés par le générateur de code. Celui-ci crée les fichiers sources en C++ au sein desquels sont intégrés les codels et le support du traceur LTTng. Enfin les fichiers sources ainsi générés et préparés sont compilés. Le processus complet est résumé dans la figure 7.1

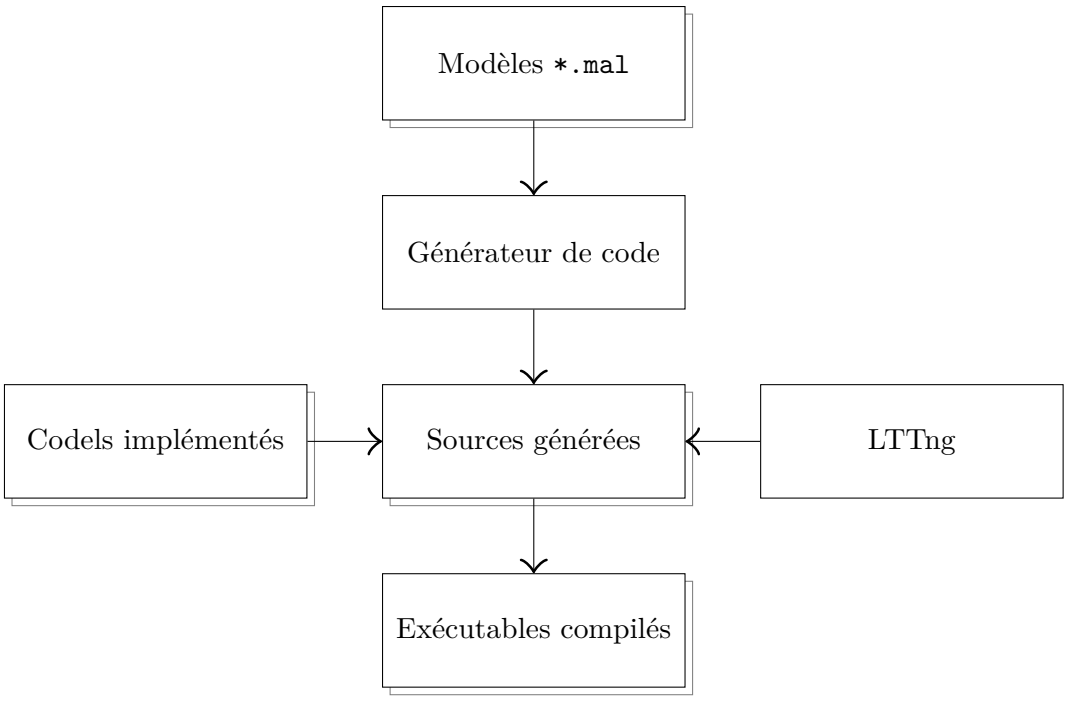


FIGURE 7.1 – Processus de génération des modèles de tâches

Ce chapitre est découpé en trois parties. La section 7.1 définit le concept de librairie et présente l'utilisation du générateur de code sur une librairie minimale. Puis la section 7.2 décrit le processus de génération de composants en étendant la librairie initiale. Enfin la section 7.3 présente la génération d'architectures ainsi que des déploiements.

7.1 Utilisation du générateur de code et concept de librairie

Le générateur de code de Mauve utilise une structure de code à base de librairies. Une librairie peut contenir tous les éléments nécessaires à la conception d'une architecture logicielle. Une librairie peut donc contenir des déclarations de codels, des définitions de types, de machines à états, de cœurs, de coquilles, de composants, d'architectures et de déploiements. De plus, une librairie peut en inclure une autre. La librairie la plus simple pouvant être générée est une librairie vide, comme le montre le listing 7.1.

Listing 7.1 – Définition de la librairie `my_lib`

```
1 library my_lib {
2 }
```

La définition d'une librairie permet de générer la structure principale d'un paquet Mauve : la génération crée l'arborescence de fichiers contenant les sources des codes et un ensemble de fichiers définissant la notion de *paquet* Mauve. Les paquets Mauve servent à gérer la composition de librairies. Les fichiers générés lors de la génération d'une librairie sont, au minimum et selon le contenu de la librairie (voir la figure 7.2) :

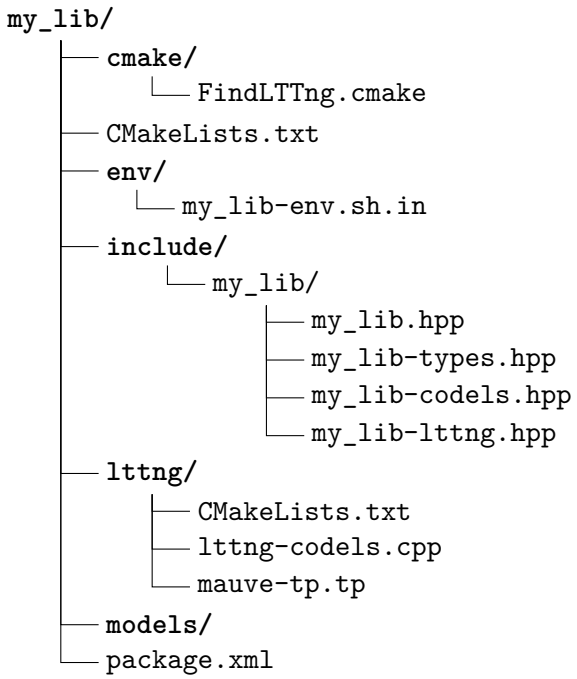


FIGURE 7.2 – Arborescence de fichiers générée par une librairie vide

— un fichier `CMakeLists.txt` définissant la configuration et la compilation du paquet ;
— un fichier *manifeste*, `package.xml`, permettant la configuration et la compilation avec `catkin` [1] ;
— un fichier *shell*, `env/librairie-env.sh.in`, définissant un ensemble de variables d'environnement nécessaires à l'utilisation du paquet généré ;
— un dossier appelé `models` contenant les fichiers de modèles Mauve de la librairie générée ;
— un dossier `include` contenant tous les fichiers d'en-tête du paquet, comme des types de données, des codels, ou des dépendances à d'autres librairies.
— enfin un dossier `lttng` associé à un fichier `cmake/FindLTTng.cmake` définissant respectivement le format des traces d'exécution et le chemin vers la librairie LTTng pour la compilation.

7.1.1 Définition d'un nouveau type de donnée

Lors du développement de logiciels, il arrive au développeur de créer des types de données non standards. Le langage de modélisation Mauve supporte la création de types de données mais ne permet pas de les définir directement. Pour créer un type de données, il faut indiquer au générateur de code le nouveau type avec le mot clé `type` au sein d'une librairie comme le montre le listing 7.2.

Listing 7.2 – Description du type non standard `A`

```
1 library my_lib {
2   type A
3 }
```

Le générateur de code initialise le fichier `include/my_lib/my_lib-types.hpp` avec la déclaration du nouveau type. Comme le langage de modélisation ne prend pas en charge le langage cible de la génération de code, il reste au développeur à définir la structure de ce nouveau type à la place de `/* your type here */` comme indiqué dans le listing 7.3.

Listing 7.3 – Code généré pour un nouveau type de données

```
1 namespace my_lib {
2 // A
3 typedef /* your type here */ A;
4 } // namespace
```

1. http://wiki.ros.org/catkin

7.1.2 Utilisation de codels existants

La philosophie de Mauve quant à la réutilisation de code est similaire à celle de ROS [Quigley et al. 2009] : les codes existants doivent pouvoir être utilisés sans que le développeur ait à les adapter ou à les réimplémenter. Pour cela, nous introduisons la notion de « code élémentaire », ou *codel* [Mallet, Pasteur et Herrb 2010] : un codel est une fonctionnalité complète et autonome, sous la forme d'une fonction.

Un codel possède des paramètres optionnels typés et orientés et une valeur de retour elle aussi typée. Les paramètres peuvent être définis comme des données d'entrée pour le codel avec le mot clé `in`, comme des données de sortie avec `out` ou comme des données bidirectionnelles avec `inout`. Afin d'intégrer un codel à une librairie Mauve, il faut utiliser le mot clé `codel`, comme présenté dans le listing 7.4.

Listing 7.4 – Déclaration d'un codel

```
1 library my_lib {
2   ...
3
4   codel first_codel(in a: A, in i: int): int
5 }
```

Le générateur de code initialise le fichier `include/my_lib/my_lib-codels.hpp` avec la déclaration du codel.

Listing 7.5 – En-tête générée pour un codel

```
1 namespace my_lib {
2 // first_codel(in a: A, in i: int): int
3 int first_codel (const ::my_lib::A& a, const int& i);
4 } // namespace
```

De plus il crée un nouveau dossier dans l'arborescence initiale : le dossier `src/codel/` contenant les définitions des codels générés, comme le montre la figure 7.3. Puis le

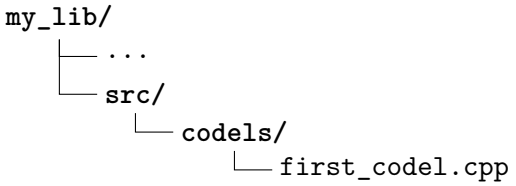


Figure 7.3 – Arborescence de fichiers ajoutée par la déclaration d'un codel

générateur crée le fichier initial de la déclaration du codel (voir le listing 7.6). Pour des raisons similaires à la création d'un type de données, le langage Mauve ne supporte pas la définition des fonctionnalités des codels dans les modèles de composants. Le développeur doit donc remplir les fichiers contenus dans `src/codel/` soit directement avec le code source de la fonctionnalité du codel, soit avec l'appel à une librairie externe.

Listing 7.6 – Code source généré pour un codel

```
#include "my_lib/my_lib.hpp"
namespace my_lib {
  // first_codel(in a: A, in i: int): int
  int
  first_codel
  (const ::my_lib::A& a, const int& i) {
    return int();
  }
} // namespace
```

Enfin le fichier `lttng/lttng-codels.cpp` est adapté pour intégrer le nouveau codel au système d'enregistrement de traces comme indiqué dans le listing 7.7.

Listing 7.7 – Code source généré pour tracer l'exécution d'un codel

```
namespace my_lib {
namespace lttng {
// first_codel(in a: A, in i: int): int
int first_codel (
const ::my_lib::A& a, const int& i) {
tracepoint(mauve_my_lib, codel, syscall(SYS_gettid), (char*)"begin", "my_lib.
    first_codel");

int r = ::my_lib::first_codel(a, i);
tracepoint(mauve_my_lib, codel, syscall(SYS_gettid), (char*)"end", "my_lib.
    first_codel");

return r;
}
}} // namespaces
```

7.2 Définition des composants

Le modèle Mauve des composants est découpé en deux parties : la coquille et le cœur, comme décrit dans la section 3.1.1. Lors de la génération de code d'un composant, pour des raisons techniques d'implémentation en C++, ces deux parties sont regroupées dans les mêmes fichiers : une en-tête et un fichier source. Le listing 7.8 montre un exemple de modélisation de composant : tout d'abord la coquille est déclarée aux lignes 4 à 8, ensuite le cœur est défini aux lignes 10 à 18 et enfin le composant est défini à partir d'une coquille et d'un cœur (ligne 20).

Listing 7.8 – Déclaration d'un composant

```
library my_lib {
  ...

  shell MyShell {
    property K: A
    input port input_port: int
```

```
7       output port output_port: int
8     }
9
10    core FooCore(MyShell) {
11      var input_value: int;
12      var output_value: int;
13      update = {
14        read(input_port, input_value);
15        output_value = input_value + 1;
16        write(output_port, output_value);
17      }
18    }
19
20    component Foo(MyShell, FooCore)
21  }
```

Le générateur de code ajoute un répertoire à l'arborescence de fichiers par composant comme indiqué dans la figure 7.4. Il ajoute dans cette nouvelle arborescence un fichier

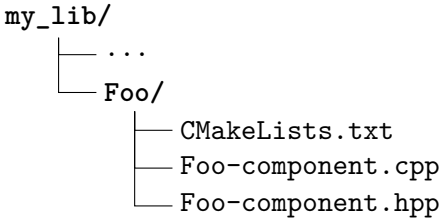


FIGURE 7.4 – Arborescence de fichiers ajoutée par la création d'un composant

`CMakeLists.txt` contenant les paramètres de compilation. Ensuite il crée et initialise les fichiers contenant les en-têtes et les sources des composants. Les listings correspondants sont disponibles en annexes, respectivement les listings B.1 et B.2. Enfin le programme périodique de chaque composant est instrumenté avec le système de trace LTTng comme indiqué aux lignes 57 à 59 et 72 à 74 du listing B.2.

7.3 Définition des architectures et des déploiements

La dernière partie concernant la génération de code consiste en la génération d'architectures logicielles puis de déploiements.

7.3.1 Déclaration des architectures

La génération d'architectures permet d'automatiser l'instanciation et la connexion de composants ainsi que la déclaration de propriétés spécifiques à cette architecture. Pour cela, le langage Mauve permet de déclarer des architectures par l'intermédiaire du

mot clé `architecture` comme présenté dans le listing 7.9. Les composants sont ensuite instanciés (lignes 5 et 6) puis connectés (ligne 7) au sein de l'architecture ainsi déclarée.

Listing 7.9 – Déclaration d'une architecture

```
1 library my_lib {
2   ...
3
4   architecture my_architecture {
5     instance foo: Foo
6     instance bar: Bar
7     connection foo.output_port -> bar.input_port
8   }
9 }
```

Le middleware cible étant Orocos, les architectures générées sont stockées sous la forme d'un *script de déploiement*, Orocos ne faisant pas la différence entre une architecture et un déploiement. Le listing 7.10 montre le script de déploiement correspondant à l'architecture présentée dans le listing 7.9. La librairie `my_lib` est tout d'abord chargée (ligne 2) pour permettre au *déployeur* d'Orocos de connaître les différents composants utilisés dans l'architecture. Ensuite les deux composants `Foo` et `Bar` sont chargés et instanciés (lignes 4 et 5). Enfin, les composants sont connectés (ligne 10) par l'intermédiaire d'une politique de connexion de type *donnée* (ligne 8).

Listing 7.10 – Script généré d'une architecture

```
1 // Import used libraries
2 import("my_lib")
3 // Create components
4 loadComponent("bar", "my_lib::Bar")
5 loadComponent("foo", "my_lib::Foo")
6 // Set component properties
7 // Connect ports
8 var ConnPolicy cp
9 cp.type = DATA
10 connect("foo.output_port", "bar.input_port", cp)
11 // Connect operations
```

7.3.2 Déclaration des déploiements

La génération de déploiements permet d'automatiser la configuration des tâches temps-réel supportant les composants. Pour cela, le langage Mauve déclare les déploiements grâce au mot clé `deployment` comme présenté dans le listing 7.11. En premier lieu, une architecture est instanciée (ligne 5) pour permettre au déploiement de connaître les composants à déployer et à configurer. Ensuite les activités de chaque composant sont spécifiées sous la forme d'une affinité, d'une priorité, d'une période et d'une échéance (lignes 6 et 12).

Listing 7.11 – Déclaration d'un déploiement

```
1  library my_lib {
2    ...
3
4    deployment my_deployment {
5      architecture my_architecture
6      activity foo {
7        affinity = 1
8        priority = 8
9        period   = 100
10       deadline = 100
11     }
12     activity bar {
13       affinity = 1
14       priority = 8
15       period   = 100
16       deadline = 100
17     }
18   }
19 }
```

Le générateur de déploiements ajoute un répertoire `scripts` au sein de l'arborescence de fichiers comme l'indique la figure 7.5. Enfin il crée le script de déploiement

```
my_lib/
├── ...
└── scripts/
    └── my_deployment.ops
```

FIGURE 7.5 – Arborescence de fichiers ajoutée par la définition d'un déploiement

`my_deployment.ops` contenant à la fois l'architecture décrite dans la section 7.3.1 et la partie spécifique au déploiement présentée dans le listing 7.12.

Listing 7.12 – Script généré d'un déploiement

```
1  // Set activities
2  foo.period = 100
3  foo.priority = 8
4  foo.deadline = 100
5  foo.affinity = 1
6  setActivity("foo", 0.1, 8, ORO_SCHED_RT)
7  foo.setCpuAffinity(1)
8  bar.period = 100
9  bar.priority = 8
10 bar.deadline = 100
11 bar.affinity = 1
12 setActivity("bar", 0.1, 8, ORO_SCHED_RT)
13 bar.setCpuAffinity(1)
14 // Configure components
15 bar.configure()
16 foo.configure()
17 // Stream ports to ROS
18 // Start components
19 bar.start()
```

```
20 foo.start()
```

7.4 Conclusion

Nous avons donc développé un logiciel de génération de code exécutable C++ exploitant les modèles de tâche à base de machines à états présentés dans le chapitre 3. Ce générateur de code permet de réduire les erreurs de modélisation en évitant au développeur de modéliser du code existant. De plus, il réduit les erreurs de mise en œuvre en évitant au développeur de développer du code à partir de modèles. Enfin, il accélère la conception d'architectures complexes en générant la partie « bas niveau » des composants et de l'architecture.

Afin de valider le fonctionnement de ces outils de modélisation, d'analyse et de génération de code, nous avons conçu une architecture logicielle à base de composants destinée à un cas d'application robotique d'exploration de zone.

Chapitre 8

Développement d'une architecture robotique modulaire

Sommaire

Résumé . *Nous avons mis en place une plate-forme robotique et un environnement d'expérimentation afin de valider le fonctionnement de nos algorithmes d'analyse d'ordonnançabilité. Pour cela, nous avons procédé en trois étapes principales :*

1. *Nous avons mis en place la plate-forme robotique en elle-même. Nous avons utilisé à la fois un environnement simulé avec le simulateur MORSE ainsi qu'un robot réel de type Pioneer P3-DX ;*
2. *Nous avons conçu une architecture simple à base de composants de type Navigation-Guidage-Contrôle ;*
3. *Nous avons exécuté et analysé l'architecture afin de valider à la fois le comportement temporel des composants de l'architecture ainsi que la validité de la méthode d'analyse.*

La méthode d'analyse présentée dans le chapitre 6 étant destinée à être principalement utilisée sur des architectures robotiques à base de composants, nous avons mis en place une plate-forme robotique ainsi que son architecture logicielle.

Ce chapitre est découpé en quatre parties. La section 8.1 décrit les différentes plates-formes expérimentales mises en place durant ces travaux de thèse. Ensuite la section 8.2 présente le type d'architecture logicielle embarquée sur le calculateur du robot. Puis la section 8.3 décrit la mission type donnée au robot ainsi que son implémentation. Enfin la section 8.4 présente les résultats de l'analyse de l'architecture logicielle embarquée fournis par la méthode décrite au chapitre 6.

8.1 Environnements d'exécution

Cette section décrit les environnements d'expérimentation utilisés durant ces travaux de thèse. Lors des développements initiaux de l'architecture de contrôle du robot ainsi que des algorithmes d'analyse d'ordonnançabilité, nous avons privilégié des expérimentations en environnement simulé, la simulation offrant des possibilités d'abstraction et des facilités de mise en œuvre que ne permettent pas les essais réels. Par contre, lors des phases finales du développement, les expérimentations réelles ont été privilégiées pour une plus grande diversité de données et une plus grande variabilité entre plusieurs exécutions « identiques ».

8.1.1 Exécution en simulation

Afin de faciliter le développement des composants formant l'architecture logicielle du robot, nous utilisons un environnement de simulation. Le simulateur utilisé est MORSE [1] (Modular OpenRobots Simulation Engine).

8.1.1.1 Présentation du simulateur MORSE

MORSE [Echeverria et al. 2012] est un simulateur générique destiné principalement à la recherche en robotique. Il est capable de simuler des environnements intérieurs ou extérieurs, petits ou grands et contenant jusqu'à des dizaines de robots autonomes.

Le simulateur est fourni avec un ensemble de capteurs usuels comme des caméras, des télémètres lasers, des GPS ou encore des odomètres. Il inclut aussi un ensemble d'actionneurs ainsi que des bases robotiques comme des quadrirotors, le Pioneer P3-DX d'Adept Mobile Robots, l'ATRV de iRobot ou le PR2 de Willow Garage.

L'affichage de la simulation est basé sur le *Game Engine* de Blender. Le Game Engine de Blender est lui-même basé sur un système OpenGL, ce qui permet le support d'options

1. https://www.openrobots.org/wiki/morse

avancées d'éclairage ou la gestion de multiples textures. De plus, la simulation du moteur physique utilise la librairie Bullet.

8.1.1.2 Présentation de l'environnement

Parmi les éléments fournis lors de l'installation de MORSE, un certain nombre d'environnements sont disponibles, à la fois d'intérieur et d'extérieur, simples ou complexes. Pour nos expérimentations, nous avons choisi un environnement plutôt simple d'intérieur dans lequel nous avons placé un robot de type Pioneer P3-DX, comme le montre la figure 8.1.

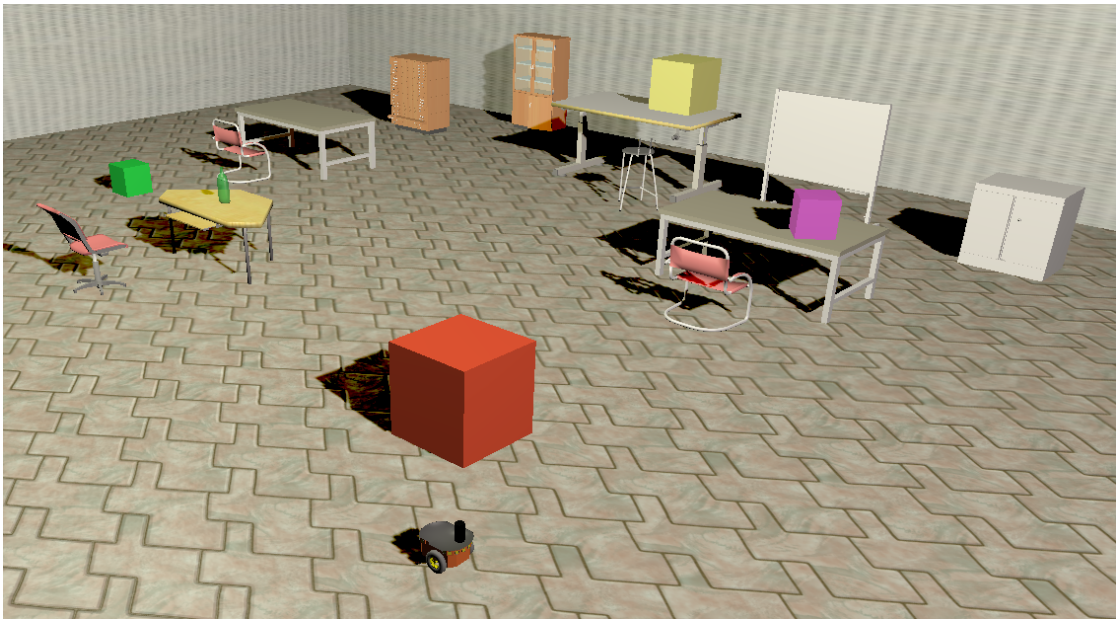


FIGURE 8.1 – Le robot dans son environnement simulé

Cet environnement, malgré sa relative simplicité, présente un certain nombre d'avantages. Il comporte de larges zones sans obstacles pour tester les capacités de mouvement du robot sans se préoccuper de collisions potentielles. Une autre partie de l'environnement est constituée d'un ensemble d'éléments statiques de mobilier dans lequel le robot doit se frayer un chemin tout en évitant les collisions. De plus quatre objets dynamiques sont présents, représentés par les cubes colorés en rouge, vert, mauve et jaune. Les objets dynamiques permettent par exemple de vérifier la détection d'éléments mobiles lors des phases de cartographie.

8.1.2 Exécution en environnement réel

Afin de valider les expérimentations effectuées en simulation, nous avons mis en place une plate-forme robotique mobile. La validation du fonctionnement de l'architecture logicielle du robot en simulation est une phase importante mais néglige une part de la complexité de l'environnement et des interactions entre le robot et l'environnement.

8.1.2.1 Présentation du matériel

Le robot utilisé lors des expérimentations est un Pioneer P3-DX d'Adept Mobile Robots (Figure 8.2a). Il est équipé d'un télémètre laser à balayage Hokuyo UTM-30LX (Figure 8.2b) ainsi que d'un calculateur embarqué Zotac Zbox CI320 nano (Figure 8.2c).

(a) Pioneer 3DX (b) Hokuyo UTM-30LX (c) Zotac Zbox CI320 nano

FIGURE 8.2 – Le robot et ses équipements

Le robot mobile Pioneer P3-DX est un robot de type unicycle capable de se déplacer à la vitesse de 1,2 m/s et de pivoter sur place. Ses moteurs et la taille de ses roues lui permettent de se déplacer dans des environnements intérieurs et en extérieur lorsque le terrain n'est pas trop accidenté. De plus il est équipé d'un ensemble de télémètres à ultrasons à l'avant afin de procurer des capacités basiques d'évitement d'obstacle.

Le télémètre laser Hokuyo UTM-30LX est un télémètre à balayage sur un arc de trois quarts de cercle. La portée du télémètre est de 30 m et peut être bridée logiciellement selon l'application. Il est de plus capable de fournir un ensemble de mesures toutes les 25 ms.

Le calculateur Zotac Zbox CI320 nano est un micro-ordinateur embarquant un processeur Intel® Celeron® N2930 à quatre cœurs cadencé à 1,83 GHz. Le système d'exploitation installé sur le calculateur est la distribution Linux Ubuntu, configurée pour fournir des capacités d'ordonnancement temps réel.

8.2 Choix d'une architecture de type Navigation-Guidage-Contrôle

Parmi les différents types d'architectures existantes, le choix pour cette thèse s'est porté sur une architecture à base de composants, modulaire, évolutive et simple à mettre en place. En avionique, et par conséquent dans les robots aériens, l'architecture de contrôle est souvent découpée en *navigation*, *guidage* et *contrôle* [Kendoul 2012 ; Kim, Sukkarieh et Wishart 2006]. Pour d'autres types de robots, cette décomposition n'est pas commune mais il est possible d'identifier une structure similaire au sein d'un grand nombre d'architectures existantes [Lacroix et al. 2002 ; Conor McGann et al. 2008]. Dans ce contexte, le choix s'est porté sur une évolution de 3T [Gat 1998] : les trois niveaux, à savoir le contrôleur, le séquenceur et le délibérateur, correspondent respectivement à des composants appelés *contrôle*, *guidage* et *navigation*.

La navigation est un composant similaire au délibérateur de 3T : il contient des algorithmes de planification. Cependant contrairement au délibérateur, la navigation ne fait que de la planification de trajectoire. Son entrée est donc constituée d'une carte de l'environnement, de la position actuelle du robot et d'un objectif à atteindre. Sa sortie consiste en un plan, ou une séquence ordonnée d'actions à effectuer pour atteindre l'objectif. Sachant que la planification de trajectoires peut être gourmande en termes de puissance de calcul, sa période d'exécution est lente par rapport au taux de variation dans l'environnement.

Le guidage est un composant analogue en terme de niveau au séquenceur de 3T. Il ne contient cependant pas de système de décision du contrôle à appliquer mais plutôt un mécanisme de suivi de trajectoires avec évitement d'obstacle réactif. Le composant de guidage prend en entrée une séquence d'actions à effectuer sous forme de trajectoire ainsi que la position du robot par rapport à la trajectoire. Sa sortie correspond aux vitesses linéaires et angulaires à appliquer par le robot afin de suivre la trajectoire. De plus des informations provenant de capteurs de proximité sont intégrées dans les algorithmes de suivi de trajectoire afin de fournir des capacités d'évitement d'obstacles réactives.

Le contrôle est un composant dont le rôle, tout comme le contrôleur de 3T, est de gérer les capacités de mouvement du robot ainsi que d'asservir les commandes appliquées aux actionneurs. L'entrée du composant de contrôle correspond à des commandes en vitesses linéaires et angulaires normalisées en unités du système international et bornées par les capacités intrinsèques du robot. Sa sortie dépend du type de robot sur lequel est exécutée l'architecture mais correspond aux commandes destinées à chaque actionneur du robot. Le contrôle contient donc un ensemble de fonctions de transfert permettant d'abstraire au guidage la gestion des commandes moteurs.

Ces trois éléments constituent le cœur de l'architecture. D'autres composants sont

ajoutés autour de ce cœur afin d'apporter plus de fonctionnalités, comme des pilotes pour communiquer avec le matériel, des algorithmes de cartographie et de localisation simultanée (SLAM), ou encore des algorithmes de traitement d'image.

8.3 Mission d'exploration

Les propriétés temps-réel de l'architecture logicielle ont été évaluées et analysées durant une mission d'exploration de zone. Le but de cette mission est d'explorer une zone inconnue de manière autonome et sûre tout en construisant une carte de l'environnement et en évitant les obstacles statiques et dynamiques.

8.3.1 Architecture de contrôle

Cette expérimentation nécessite un ensemble de fonctionnalités autour de l'architecture Navigation-Guidage-Contrôle de base : un algorithme le cartographie, des fonctions d'évitement d'obstacle, un algorithme d'exploration à partir d'une carte ainsi qu'un planificateur. De plus, les pilotes des organes matériels du robot sont implémentés, tels que le pilote du télémètre laser et du contrôleur des moteurs. Chacune de ces fonctionnalités sont implémentées au sein de différents composants comme le montre la figure 8.3

Les fonctions de bas niveau, les pilotes, représentés en jaune, se trouvent dans les composants suivants :

— *hokuyo* : il s'agit du pilote du télémètre laser. Il contient les fonctions d'initialisation et de configuration du télémètre et fournit périodiquement une mesure de la nappe laser ;

— *p3dx* : il s'agit du pilote du robot. Ce composant initialise le protocole de communication entre le calculateur et le contrôleur interne du robot. Il permet ensuite d'envoyer de manière périodique des commandes en vitesse linéaires et angulaires et de recevoir des données de position du robot calculée à partir de ses encodeurs.

Les fonctions de localisation, en rouge, sont attachées à deux composants :

— *SLAM* : ce composant sert d'interface à l'algorithme GMapping [Grisetti, Stachniss et Burgard 2007]. Il s'agit d'un algorithme de localisation et de cartographie simultanée. Il utilise les nappes laser ainsi que la pose estimée par les encodeurs du robot afin de construire une carte de l'environnement. Il calcule aussi une erreur d'estimation de la pose fournie par les encodeurs du robot ;

— *correcteur de pose* : ce composant sert à calculer la pose corrigée du robot. L'estimation de la pose du robot par ses encodeurs est peu précise mais est

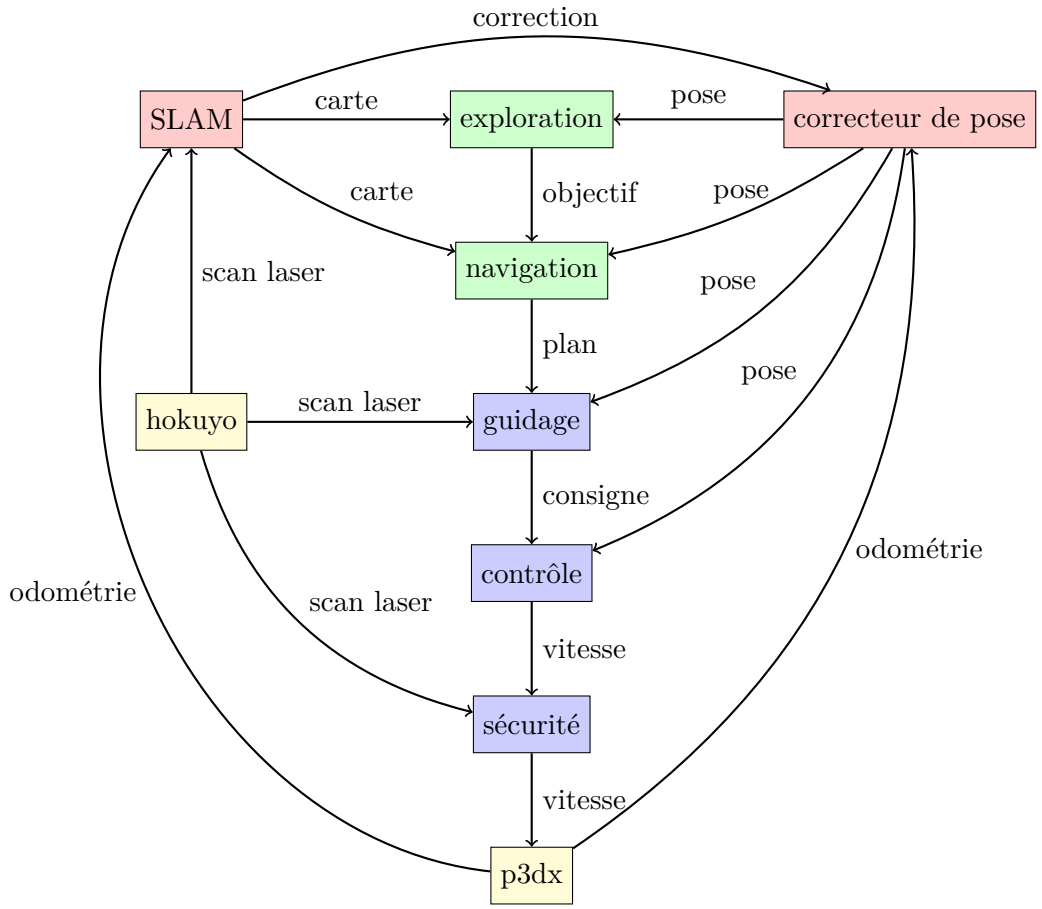


FIGURE 8.3 – L'architecture d'exploration du robot

disponible à une fréquence élevée alors que celle fournie par GMapping est précise mais les calculs pour l'obtenir sont longs et complexes. Ce composant utilise donc les données des encodeurs du robot et applique la correction de l'erreur de pose fournie par le SLAM.

Le robot est contrôlé par un ensemble de composants de plus ou moins haut niveau, en bleu et en vert, pour sélectionner des zones à explorer, calculer un plan de navigation pour y accéder et exécuter ce plan. Les composants associés sont *exploration*, *navigation*, *guidage*, *contrôle* et *sécurité* :

— *exploration* : le rôle de ce composant est de sélectionner sur la carte construite par le SLAM un point non encore cartographié et *a priori* accessible. Le point sélectionné est situé à la frontière entre la carte connue et les zones non encore détectées. De plus les point contenus au sein de zones inaccessibles ne sont pas sélectionnés ;

— *navigation* : il s'agit du planificateur de trajectoires de l'architecture. Ce composant utilise donc la pose courante du robot, l'objectif à atteindre défini par le composant d'exploration et la carte de l'environnement pour calculer un chemin faisable entre les deux points avec l'algorithme A* [Hart, Nilsson et Raphael 1968]. Ce chemin est filtré pour en extraire un ensemble de points de passage accessibles par le robot ;
— *guidage* : ce composant a pour but de guider le robot le long de la trajectoire calculée par le composant de navigation. Pour cela, il utilise séquentiellement les points de passages du plan et vérifie grâce aux nappes laser si la ligne entre la position courante du robot et le point de passage suivant est libre d'obstacles. L'évitement d'obstacles ainsi réalisé est principalement destiné aux obstacles dynamique ne pouvant pas être pris en compte lors de la planification initiale ;
— *contrôle* : ce composant est le siège de la commande du robot. Pour cela il implémente un contrôleur de type proportionnel-integral-dérivé afin d'orienter le robot vers son objectif local fourni par le guidage ;
— *sécurité* : ce composant possède un rôle particulier dans l'architecture. Contrairement aux autres, il n'ajoute pas de fonctionnalités : il sert uniquement d'« arrêt d'urgence ». Lorsqu'un obstacle est détecté par le laser dans la direction d'avancement du robot à une distance inférieure à une marge de sécurité, il empêche le robot d'avancer dans cette direction.

8.3.2 Spécification Mauve de l'architecture

L'architecture précédemment décrite est constituée de neuf composants distincts, décrits sous la forme de quatorze modèles Mauve. La plupart des composants sont séparés en deux modèles : le premier contient la déclaration des fonctionnalités du composant sous la forme de codels. Le second modèle décrit la structure des composants et la logique nécessaire au bon fonctionnement des codels. Bien que la modélisation de composants ne nécessite pas une telle séparation, celle-ci permet de réutiliser plus facilement différents codels au sein de plusieurs composants.

Parmi les neuf composants présents au sein de cette architecture, six d'entre eux contiennent des machines à états : le contrôle, l'exploration, la cartographie, le guidage, la navigation et la sécurité. Les trois autres composants possèdent une exécution linéaire et ne contiennent donc pas de machine à états.

Au global, les modèles de l'architecture complète se résument en 1 074 lignes de code et les codels nécessaires au fonctionnement des composants sont constitués de 1 479 lignes de code. Le générateur de code utilise les modèles et les codels pour générer les sources des composants contenant 5 658 lignes auxquelles s'ajoutent les fichiers permettant la compilation et le déploiement de l'architecture. Les listings complets des modèles Mauve

des composants de cette architecture sont fournis dans l'annexe C.

8.4 Analyse d'ordonnançabilité

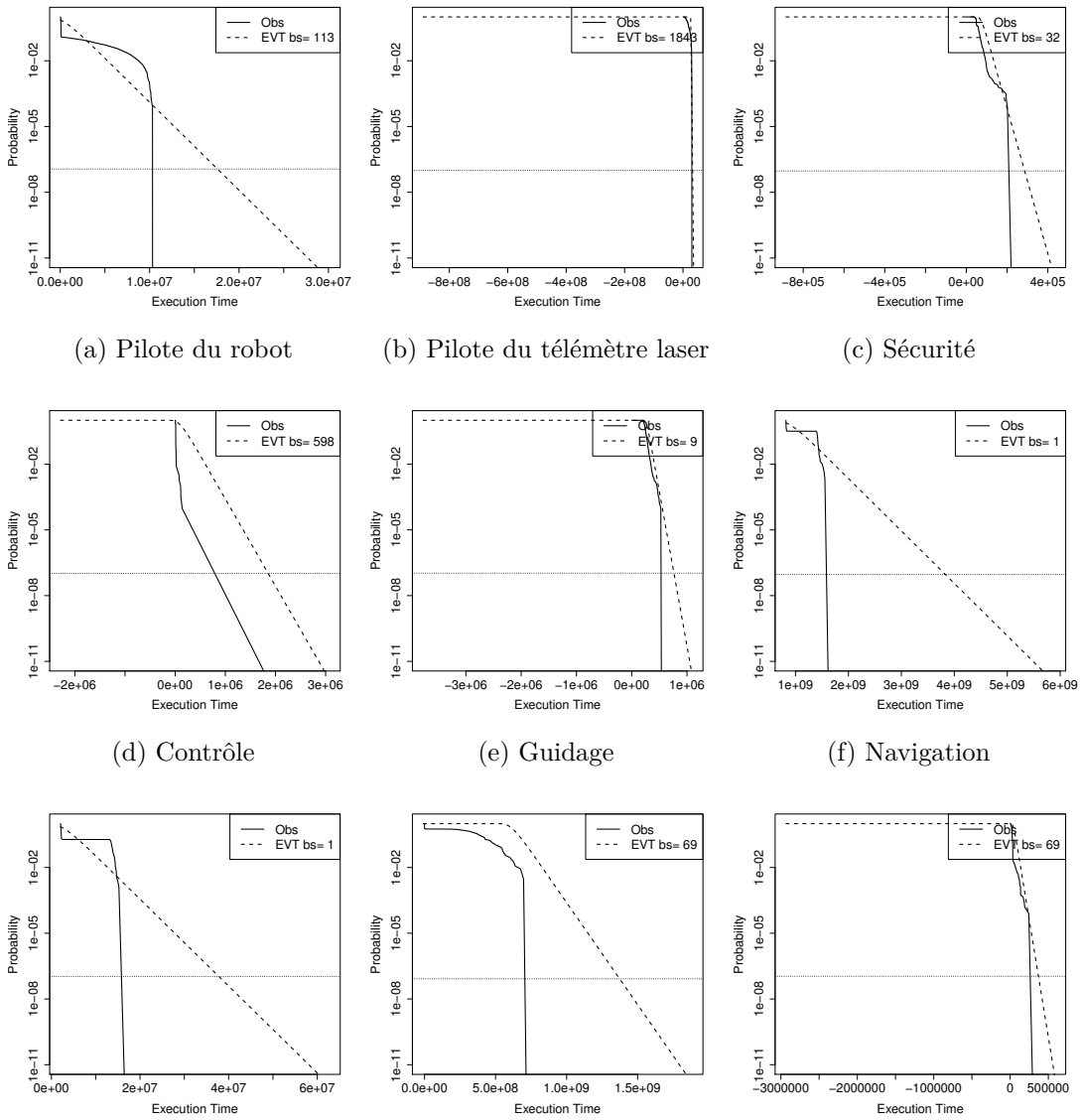
L'objectif principal de la mise en place d'une plate-forme d'expérimentation robotique est de vérifier la validité de la méthode d'analyse présentée dans le chapitre 6 sur un cas d'application réel. Pour cela, nous avons procédé en trois étapes : nous avons exécuté l'architecture logicielle d'exploration sur la plate-forme robotique afin de collecter des traces d'exécution ; nous avons ensuite estimé les pires temps d'exécution de chaque fonction ; enfin nous avons calculé les pires temps de réponse des composants pour conclure sur l'ordonnançabilité du système.

8.4.1 Estimation des pires temps d'exécution

Nous avons généré le code Orocos des composants de l'architecture décrite dans la section 8.3 à partir de leurs modèles Mauve afin d'en collecter des traces d'exécution. Bien que les temps d'exécution des composants et de leurs fonctions dépendent de la cible matérielle sur laquelle ils s'exécutent, l'indépendance temporelle entre composants permet de collecter des traces d'exécution indépendamment de l'architecture dans laquelle ils se trouvent. L'estimation des pires temps d'exécution des composants a donc été effectuée lors de plusieurs expérimentations :

— contrôle automatique : lors de cette expérimentation, seuls les composants liés au contrôle du robot (les composants bleus et jaunes de la figure 8.3) étaient actifs. Nous avons envoyé manuellement des objectifs au guidage pendant 21 minutes afin de collecter suffisamment de données pour estimer les pires temps d'exécution des fonctions de ces composants ;
— cartographie : cette expérimentation consiste à exécuter les composants liés à la localisation et cartographie simultanée (les composants rouges et jaune de la figure 8.3). Le robot a été déplacé manuellement pendant environ une heure ;
— planification : lors de cette expérimentation, seuls les composants de navigation et d'exploration (en vert sur la figure 8.3) ont étés utilisés. Pour permettre à ces deux composants de fonctionner, nous leur avons fourni la carte construite lors de l'expérimentation précédente et changé régulièrement le point de départ simulant la position du robot pendant 50 minutes.

La figure 8.4 regroupe les distributions cumulées inverses des mesures de temps d'exécution des codels principaux des composants impliqués dans l'architecture de contrôle du robot ainsi que les distributions de probabilité calculées avec la théorie des valeurs extrêmes. Les données utilisées ici proviennent des différentes expérimentations effectuées.



(a) Pilote du robot (b) Pilote du télémètre laser (c) Sécurité

(d) Contrôle (e) Guidage (f) Navigation

(g) Exploration (h) SLAM (i) Correcteur de pose

FIGURE 8.4 – Distributions cumulées inverses et application de la théorie des valeurs extrêmes sur les codels principaux des composants impliqués dans l'architecture de contrôle du robot

Les temps d'exécution des codels sont déterminés lorsque la distribution calculée avec la théorie des valeurs extrêmes croise une certaine probabilité. Dans nos expérimentations,

cette probabilité a été choisie à $1 \cdot 10^{-7}$. Les pires temps d'exécution ainsi collectés sont résumés dans la table 8.1. Le temps d'exécution de tous les codels impliqués au sein des

TABLE 8.1 – Résultats de l'analyse de pire temps d'exécution de quelques codels

composant (codel)	WCET (ms)
Pilote du robot (aria_loop_once)	17,9
Pilote du télémètre (get_scan)	36,6
Sécurité (hitPrevention)	0,3
Contrôle (pid_control)	1,9
Guidage (nextPoint)	0,8
Correcteur de pose (pose_correction)	0,4
Navigation (wavefront)	3 816,3
Exploration (frontier)	38,0
SLAM (addScan)	1 378,9

composants sont regroupés dans l'annexe D.

8.4.2 Calcul des pires temps de réponse

L'analyse de pire temps de réponse utilise les modèles de composants et d'architecture ainsi que les pires temps d'exécution des fonctions et de la glue pour calculer le pire temps de réponse de chaque composant. La spécification du déploiement est décrite dans la table 8.2.

TABLE 8.2 – Spécification du déploiement

composant	période (ms)	échéance (ms)	priorité	affinité
p3dx	100	100	10	1
sécurité	100	100	9	1
hokuyo	250	250	8	1
contrôle	250	250	7	1
correcteur de pose	250	250	6	1
guidage	500	500	5	1
navigation	7 000	7 000	4	1
exploration	30 000	30 000	2	1
SLAM	4 000	4 000	3	2

L'analyse de pire temps de réponse des composants de l'architecture est calculée avec l'équation (6.27) et des couples $\langle \overline{C}, p \rangle$ avec une confiance de 10^{-7}. Ces résultats sont présentés dans la table 8.3.

TABLE 8.3 – Résultats de l'analyse de pire temps de réponse

composant	WCRT (ms)	échéance (ms)	charge processeur (%)
p3dx	23	100	23
sécurité	26	100	26
hokuyo	65	250	41,6
contrôle	68	250	42,8
correcteur de pose	70	250	43,6
guidage	72	500	44
navigation	6 874	7 000	98,6
exploration	6 942	30 000	98,7
SLAM	1 556	4 000	38,9

La charge processeur U_i indique la pire charge processeur cumulée pour chaque composant i : $U_i = \sum_{j \leq hp(i)} \frac{\mathcal{C}_j}{T_i}$. L'architecture est ordonnançable avec une charge processeur maximale de 98,7 % sur le cœur №1, c'est à dire sur le cœur exécutant tous les composants de l'architecture sauf la cartographie. Le composant de cartographie quant à lui utilise au maximum 38,9 % du cœur du processeur sur lequel il s'exécute. De plus, la charge processeur passée dans la glue, c'est à dire dans le middleware et la coquille des composants, est d'environ 3 % dans le pire cas.

8.4.3 Validation expérimentale

L'analyse de pire temps de réponse de la section précédente démontre que les contraintes temps-réel de chaque composant sont respectées lors de l'exécution. Sachant que l'analyse a été effectuée à partir de plusieurs expérimentations minimales dans lesquelles l'architecture n'était pas déployée en entier, nous avons comparé les traces d'une expérimentation complète avec les pires temps de réponse de l'analyse. Nous avons donc déployé l'architecture complète pour une mission d'exploration de 10 minutes. La figure 8.5 montre la trajectoire du robot ainsi que la zone explorée.

La table 8.4 montre la comparaison entre les temps de réponse maximaux observés pendant la mission d'exploration et l'estimation de ces pires temps de réponse donnés par l'analyse précédente.

Nous pouvons noter que les temps de réponses maximaux mesurés sont bien en deçà des pires temps de réponse calculés, démontrant que l'estimation est sûre. La différence est cependant assez grande impliquant une analyse trop pessimiste : le pire cas d'exécution est calculé lorsque les composants de priorité élevée préemptent les autres composants. Ce phénomène n'arrive en pratique que peu souvent mais nous ne pouvons pas relaxer cette hypothèse puisque nous n'avons pas d'informations sur la synchronisation des tâches. Ce

FIGURE 8.5 – Carte explorée avec la trajectoire du robot

TABLE 8.4 – Comparaison de résultats expérimentaux

composant	mesure (ms)	WCRT (ms)	échéance (ms)
p3dx	11	23	100
sécurité	1	26	100
hokuyo	31	65	250
contrôle	3	68	250
correcteur de pose	1	70	250
guidage	1	72	500
navigation	1 619	6 874	7 000
exploration	18	6 942	30 000
SLAM	856	1 556	4 000

point constitue une des perspectives discutées dans la conclusion. Néanmoins, même si l'analyse est pessimiste, elle donne des résultats d'ordonnançabilité sûrs.

8.5 Conclusion

Cette étape d'expérimentation nous a permis de mettre en application la méthodologie complète de Mauve qui consiste à modéliser, déployer et analyser des architectures robotiques. Nous avons donc confiance dans le fait que le déploiement et l'exécution de l'architecture soient conformes à sa spécification. De plus, l'analyse temps-réel de

l'architecture correspond au système réel grâce à l'utilisation du générateur de code.

Pour cette phase de validation expérimentale, nous avons mis en place un environnement d'expérimentation à la fois en simulation grâce au simulateur MORSE et en environnement réel. Pour cela, nous avons développé une architecture logicielle d'exploration de zone à base de tâches de type Navigation-Guidage-Contrôle. Ensuite l'exécution de cette architecture par morceaux a permis de collecter des traces d'exécution. Enfin nous avons analysé ces traces d'exécution et prouvé l'ordonnançabilité du système complet.

Conclusion et perspectives

Conclusion

Ces travaux de thèse sont issus de l'observation suivante : les systèmes temps-réel sont généralement construits autour de modèles d'exécution simples mais pour lesquels il existe des méthodes d'analyses adaptées. Cependant, les architectures robotiques sont souvent constituées de composants possédant des modèles d'exécution détaillés et complexes pour lesquels il n'existe pas de méthode d'analyse. De plus les outils de conception d'architectures logicielles destinés à la robotique autonome ne prennent que très rarement en compte les aspects temporels et les contraintes temps-réel dans la phase de conception.

L'objectif de ces travaux de thèse est donc d'adapter les méthodes d'analyse temps-réel au modèle d'exécution d'architectures robotiques. Pour cela, nous avons procédé en quatre étapes principales. Nous avons d'abord conçu un langage de modélisation adapté à la fois aux besoins de la robotique mais possédant aussi l'expressivité nécessaire à la caractérisation de systèmes temps-réel. La seconde étape consiste à analyser l'architecture ainsi modélisée afin de déterminer de manière précise et rapide le pire temps d'exécution puis le pire temps de réponse des différents composants prenant part au fonctionnement de l'architecture. Ensuite nous avons développé un générateur de code utilisant les modèles de composants afin de générer du code exécutable par le calculateur du robot tout en garantissant la validité des résultats d'ordonnançabilité lors de l'exécution réelle. Enfin nous avons procédé à une validation expérimentale en exécutant une architecture robotique d'exploration sur un robot mobile en conditions réelles.

Modélisation des composants

Afin de modéliser à la fois des architectures logicielles pour la robotique autonome et des systèmes temps-réel, nous avons développé le langage de modélisation Mauve. Celui-ci permet de modéliser des architectures robotiques à base de composants et leurs déploiements. Ces travaux ont fait l'œuvre d'une communication au congrès SIMPAR [Gobillot, Lesire et Doose 2014].

Langage de modélisation : Un composant est constitué par deux éléments : une *coquille* qui fournit l'interface du composant et un *cœur* qui, associé à une coquille, définit son comportement. Le comportement du composant peut être spécifié sous la forme d'une fonction unique ou sous la forme d'une machine à états. Une architecture permet

d'assembler, d'instancier et de configurer les composants précédemment définis. De plus les architectures définissent les connexions entre composants. Un déploiement permet d'instancier une architecture et de configurer les paramètres d'exécution des composants tels que leur période, leur échéance ou leur affinité.

Modèle de tâche : Les composants sont associés à des tâches périodiques pour pouvoir s'exécuter. Nous avons défini un modèle de tâche dérivé du modèle canonique constitué d'une période, d'une échéance et d'un temps d'exécution. La différence principale entre ces deux modèles est la notion de temps d'exécution des tâches : dans le modèle canonique, il s'agit d'une valeur unique et spécifique à chaque tâche. Dans notre modèle, le temps d'exécution est extrait à partir des machines à états des composants.

Analyse temps-réel

Sachant que les architectures robotiques contiennent souvent des composants formés autour de machines à états, nous avons choisi de prendre en compte ces machines à états lors des analyses d'ordonnançabilité. Pour cela, nous avons développé une méthode de calcul du temps d'exécution de composants contenant des machines à états afin de déterminer le pire temps de réponse de ces composants au sein d'une architecture. Ces travaux ont fait l'œuvre d'une communication au congrès ETFA [Gobillot, Doose, Lesire et al. 2015].

Machines à états périodiques : Le modèle de composant définit que leur comportement est spécifié par des machines à états. Pour pouvoir utiliser ces machines à états au sein des analyses d'ordonnançabilité, nous définissons le concept de machine à états périodique. Une machine à états périodique définit le comportement temporel de la machine à états du composant.

Pire temps d'exécution : Les machines à états périodiques sont représentées sous la forme de graphes dont les états sont les nœuds et les transitions sont les arcs. Cette représentation permet aisément de calculer les fonctions de requête de chaque machine à états et d'en déterminer une borne supérieure. La borne supérieure des fonctions de requête représente le pire temps d'exécution cumulé du composant sur plusieurs itérations successives.

Pire temps de réponse : Le calcul du pire temps de réponse est effectué en utilisant les bornes supérieures des fonctions de requête des composants en lieu et place

des temps d'exécution des tâches. Cela permet de conserver les méthodes d'analyses d'ordonnançabilité existantes tout en augmentant leur précision.

Performances de l'analyse : L'estimation des performances de cette méthode d'analyse de situe sur deux niveaux : l'augmentation de complexité liée à un traitement plus fin des composants en exploitant leurs machines à états et le gain de précision du calcul de pire temps de réponse. Au global la complexité calculatoire augmente peu : elle augmente linéairement avec le nombre d'états de la machine à états, le nombre de transitions et la durée de la période d'étude. Le gain de précision dépend de deux éléments : la variabilité de temps d'exécution de chaque fonction exécutée par la machine à états et aussi la durée de la période d'étude.

Génération de code

Nous avons mis en place un logiciel de génération de code à partir de notre langage de modélisation pour réduire les différences de spécification entre modèle et implémentation et réduire les temps de développement d'applications robotiques. Le code de l'architecture robotique est généré à partir de la spécification des modèles de composants, d'architectures et de déploiements. Cela permet de garantir une différence minimale entre les temps de réponse déterminés à partir des modèles et les temps de réponse observables lors d'une exécution réelle. De plus en réduisant la quantité de code à développer, les cycles de développement sont plus courts.

Validation expérimentale

Enfin, dans le but de vérifier l'applicabilité de nos outils de modélisation, d'analyse et de génération de code, nous avons développé une architecture robotique à base de composants. Elle est de type Navigation-Guidage-Contrôle et permet à un robot mobile d'explorer automatiquement une zone tout en évitant les éventuels obstacles statiques et dynamiques. Nous avons ensuite généré le code de cette architecture puis exécuté par morceaux afin d'en collecter les temps d'exécution. Enfin nous avons validé l'ordonnançabilité de l'architecture complète et exécuté une mission d'exploration. Ces travaux ont fait l'œuvre de deux communications à l'atelier SDIR de la conférence ICRA [Gobillot, Lesire et Doose 2013] et aux journées francophones CAR [Gobillot, Doose, Grand et al. 2015].

Perspectives

Sur la base de ces travaux, plusieurs évolutions peuvent être envisagées. Il en ressort deux catégories principales : la première consiste à étendre le spectre et la précision des résultats d'analyse tout en conservant la même méthodologie. La seconde catégorie consiste à enrichir le modèle d'exécution.

Extension des analyses

Afin d'améliorer la précision de l'estimation du pire temps de réponse des composants tout en conservant la même méthodologie, nous envisageons trois évolutions possibles : la première se focalise sur les conditions dans lesquelles les temps d'exécution probabilistes sont déterminés. La seconde évolution consiste à exploiter l'aspect probabiliste des temps d'exécution afin de fournir des temps de réponses probabilistes. Enfin la troisième évolution augmente le spectre d'analyse temps-réel en diversifiant les mécanismes d'ordonnancement.

Applicabilité des temps d'exécution probabilistes : Les temps d'exécution probabilistes des fonctions utilisées par les composants sont calculés à partir de mesures effectuées durant une ou plusieurs exécutions de ces fonctions. Cependant, les conditions dans lesquelles sont collectées ces mesures varient d'une exécution à une autre et peuvent induire une imprécision par rapport aux conditions nominales d'utilisation.

La précision des analyses statiques provient, entre autres, d'une spécification précise des conditions d'utilisation permettant par exemple de borner le nombre d'itérations de boucles. Dans le cas d'analyses probabilistes à base de mesures, ces conditions ne sont pas spécifiées et ne peuvent pas être connues à partir de données issues d'expérimentations réelles. Par exemple, dans le cas d'un algorithme de cartographie, la taille de la carte générée lors d'une mission n'est ni connue ni bornée, par conséquent les mesures de temps d'exécution n'ont aucun lien explicite avec les conditions d'exécution.

Pour pallier à ce manque de précision, nous proposons de créer des cas de tests abstraits représentatifs de conditions réelles. Ces tests doivent être paramétrables pour chaque codel afin que l'utilisateur puisse spécifier avec précision les conditions d'utilisation et ainsi collecter des mesures représentatives des conditions nominales.

Calcul de pire temps de réponse probabilistes : Les temps de réponse calculés par Mauve exploitent sans distinctions des temps d'exécution statiques et probabilistes puisque la méthode d'analyse ne prend qu'une valeur seuillée des temps d'exécution probabilistes.

Cette procédure possède l'avantage de pouvoir exploiter des temps d'exécution provenant de différentes sources au prix d'une perte d'information.

Le calcul de temps de réponse probabiliste [Lu et al. 2012] permet de connaître la probabilité qu'une tâche dépasse son échéance. Cette information est intéressante pour des systèmes temps-réel souples ou hybrides en donnant le niveau de dégradation de la qualité de service.

Pour permettre à Mauve de fournir des temps de réponse probabilistes, il faut utiliser les distributions de probabilité représentant les temps d'exécution des codels dans le calcul des traces, puis déterminer une borne supérieure probabiliste des traces et enfin exploiter ces bornes supérieures dans des analyses d'ordonnançabilité.

Ordonnanceurs à priorités dynamiques : Le calcul du pire temps de réponse de tâches logicielles permettant de déterminer l'ordonnançabilité d'un système de tâches dépend du temps d'exécution des tâches mais aussi des interactions entre ces tâches selon un mécanisme de priorités. Les priorités des tâches sont décidées par l'ordonnanceur en fonction de divers critères comme la période ou l'échéance des tâches.

Dans le cadre de ces travaux, nous nous sommes focalisés sur l'analyse de systèmes ordonnancés par des ordonnanceurs à priorités fixes. Cependant, l'utilisation d'ordonnanceurs à priorités dynamiques est commune et permet une meilleure utilisation processeur.

Pour permettre à Mauve d'analyser des architectures logicielles ordonnancées par des ordonnanceurs à priorités dynamiques, il faut implémenter au sein de l'algorithme permettant de déterminer le pire temps de réponse un mécanisme de gestion des priorités. De plus le générateur de code doit être adapté afin d'indiquer au déploiement le protocole d'ordonnancement des tâches.

Évolution du modèle d'exécution

La méthodologie de Mauve a été conçue pour fournir rapidement et facilement des résultats d'ordonnançabilité fiables à l'utilisateur. Cependant, pour augmenter la précision des analyses de Mauve, cette méthodologie doit évoluer. Pour cela, nous proposons plusieurs évolutions : déterminer l'état initial des machines à états des composants, faire varier les paramètres des tâches durant l'exécution ou encore rapprocher Mauve de langages de modélisation existants.

État initial des machines à états périodiques : La détermination de pire temps d'exécution des composants se base sur un calcul de fonction de requête à partir du

graphe que représente les machines à états. Notre modèle d'exécution ne permet de définir l'état initial de la machine à états que pour la première exécution du composant. Le calcul du pire temps de réponse est déterminé à partir d'un instant critique, défini comme la date à laquelle tous les composants de l'architecture commencent leur exécution. L'état de la machine à états à cet instant est donc inconnu.

Ne pas connaître l'état initial de la machine à états d'un composant génère une imprécision dans le calcul de sa fonction de requête : pour que cette fonction de requête ne soit jamais optimiste, sa première valeur correspond au pire temps d'exécution de la machine à états. Afin de réduire le pessimisme de la fonction de requête des composants, il serait intéressant de déterminer avec précision les états initiaux de chaque machine à états impliquées au sein d'une architecture logicielle afin de calculer précisément le pire temps de réponse de chaque composant.

Pour cela, il est nécessaire de connaître la date de l'instant critique sachant les conditions initiales de l'exécution. Cela permet de déterminer un ensemble limité d'états par machine à états au moment de l'instant critique et ainsi augmenter la précision du calcul des pires temps de réponse.

Paramètres des tâches variables : Lors de l'exécution de composants contenant des machines à états, certains états d'un composant peuvent contenir des fonctions critiques au bon fonctionnement de l'architecture et nécessiter une fréquence d'activation différente. Par exemple, un composant de traitement d'image devant localiser un objet dans une scène puis le suivre afin de fournir les coordonnées de cet objet ne possède pas les mêmes contraintes d'exécution lors de la détection ou lors du suivi.

Lorsque ces fonctions sont implémentées au sein d'une même machine à états, elles sont exécutées avec la même priorité et la même période. Sachant que la détection d'objet dans une image est plus coûteuse en temps de calcul que le suivi, ce dernier peut être exécuté plus fréquemment.

Afin de permettre à Mauve de traiter des composants possédant des paramètres variables, il faut adapter la méthode de calcul des traces afin de prendre en compte la périodicité spécifique de chaque état. Il faut aussi annoter les états contenus dans les traces pour permettre à l'algorithme de calcul de pire temps de réponse de connaître la priorité effective des tâches. De plus, le générateur de code doit produire les informations nécessaires au système d'exploitation pour gérer les périodes et les priorités variables.

Annexe A

Grammaire du langage Mauve

Le but de cette annexe est de présenter la grammaire complète du langage de modélisation Mauve. Deux partie principales composent ce langage : le langage de modélisation permettant de définir les éléments constitutifs d'une architecture robotique et le langage d'expression fournissant un support basique d'instructions et d'expressions au générateur de code.

A.1 Langage de modélisation

A.1.1 Définition de librairies

A.1.1.1 Exemple

```
1 library my_library {
2   // library elements
3 }
```

A.1.1.2 Syntaxe

```
1 <library>          <- 'library' <identifier> '{' <library_element>* '}'
2 <library_element> <- <use> | <type_def> | <codel_def> | <shell_def> | <
      core_def> | <component_def> | <architecture_def> | <deployment_def>
```

A.1.2 Dépendances à d'autres librairies

A.1.2.1 Exemple

```
1 library my_library {
2   use another_library
3   use yetanother_library
4   // ...
5 }
```

A.1.2.2 Syntaxe

```
1 <use> <- 'use' <identifier>
```

A.1.3 Définition de types

A.1.3.1 Exemple

```
1 library my_library {
2   // define a new type
3   type NewType
4
5   // define a supertype and its subtype
6   type SuperType
7   type SubType extends SuperType
8 }
```

A.1.3.2 Syntaxe

```
1 <type_def> <- 'type' <identifier> ('extends' <identifier>)?
```

A.1.4 Définition de codels

A.1.4.1 Exemple

```
1 library my_library {
2   codel my_codel(in i: int, inout n: NewType, out f: float): boolean
3   codel anotherone(): void
4 }
```

A.1.4.2 Syntaxe

```
1 <code_def>        <- 'codel' <identifier> '(' <parameters_list>? ')' ':' <
      identifier>
2 <parameters_list> <- <parameter_def> | (<parameter_def> , <parameters_list>)
3 <parameter_def>   <- (<access>)? <identifier> ':' <identifier>
4 <access>          <- 'in' | 'out' | 'inout'
```

A.1.5 Définition de coquille de composants

A.1.5.1 Exemple

```
shell MyShell {
  property prop: float
  input port input: MyType
  output port: YourType
  provide my_operation(): boolean
  require your_operation(i: int, d: double): SubType
}
```

A.1.5.2 Syntaxe

```
<shell_def>      <- 'shell' <identifier> ('extends' <identifier>)? '{' <
    shell_element>* '}'
<shell_element> <- <property_def>  | <port_def> | <provided_def>  | <
    required_def>
<property_def>  <- 'property' <identifier> ':' <identifier> ('=' <expression>)
    ?
<port_def>      <- ('input' | 'output') 'port' <identifier> ':' <identifier>
<provided_def>  <- 'provide' <identifier> '(' <argument_list>? ')' ':' <
    identifier>
<required_def>  <- 'require' <identifier> '(' <argument_list>? ')' ':' <
    identifier>
<argument_list> <- <argument_def> | (<argument_def>, <argument_list>)
<argument_def>  <- <identifier> ':' <identifier>
```

A.1.6 Définition de cœur de composants

A.1.6.1 Exemple

```
core MyCore (MyShell) {
  // variables definition
  var i: int = 1;
  handler h: your_operation
  // Operation definition
  provide my_op(i: int, j: int) = {
    return i + j;
  }
  // Hooks
  configure { /* ... */ }
  start { /* ... */ }
  stop { /* ... */ }
  cleanup { /* ... */ }
  update { /* ... */ }
}
```

A.1.6.2 Syntaxe

```
<core_def>       <- 'core' <identifier> '(' <shell_ref> ')' '{' <core_element
    >* '}'
```

```
<core_element>   <- <variable_def> | <handler_def> | <operation_def> | <hook>
<variable_def>   <- 'var' <identifier> ':' <identifier> ('=' <expression>)? ';'
<handler_def>    <- 'handler' <identifier> ':' <identifier>
<operation_def>  <- 'provide' <identifier> '(' <argument_list>? ')' '=' <program>
<hook>           <- <configure_hook> | <start_hook> | <update_hook> | <stop_hook> | <cleanup_hook> | <statemachine_def>
<configure_hook> <- 'configure' '=' <program>
<start_hook>     <- 'start' '=' <program>
<stop_hook>      <- 'stop' '=' <program>
<cleanup_hook>   <- 'cleanup' '=' <program>
<update_hook>    <- 'update' '=' <program>
```

A.1.7 Définition de machines à états

A.1.7.1 Exemple

```
statemachine {
  var v: int
  var w: int
  initial state Init {
    entry = { v = 0; }
    run = { v = v + 1; }
    transition t1 if (v > 5) to Final
  }
  state Final {
    entry = { w = 10; }
    exit = { w = 0; }
    transition t2 to Init { v = w; }
  }
}
```

A.1.7.2 Syntaxe

```
<statemachine_def> <- 'statemachine' '{' (<variable_def> | <state_def>)* '}'
<state_def>        <- ('initial')? 'state' <identifier> '{' <action_def>* <transition_def>* '}'
<action_def>       <- <entry_hook> | <run_hook> | <exit_hook> | <handle_hook>
<entry_hook>       <- 'entry' '=' <program>
<run_hook>         <- 'run' '=' <program>
<exit_hook>        <- 'exit' '=' <program>
<handle_hook>      <- 'handle' '=' <program>
<transition_def>   <- 'transition' <identifier> (<transition_guard>)? 'to' <identifier> (<program>)?
<transition_guard> <- 'if' <expression>
```

A.1.8 Définition de composants

A.1.8.1 Exemple

```
component MyComponent (MyShell, MyCore)
```

A.1.8.2 Syntaxe

```
<component_def> <- 'component' <identifier> '(' <identifier> ',' <identifier>
    ')'
```

A.1.9 Définition d'architectures

A.1.9.1 Exemple

```
architecture Architecture {
  // instances
  instance inst_1: MyComponent
  instance inst_2: MyComponent
  // Port connections
  connection inst_1.out ->      inst_2.in
  connection inst_2.out ->[10] inst_1.in
  // Operation connection
  operation inst_1.f -> inst_2.g
  // Property
  property inst_1.prop = 3
}
```

A.1.9.2 Syntaxe

```
<architecture_def>      <- 'architecture' <identifier> '{' <
    architecture_element>* '}'
<architecture_element> <- <instance_def> | <port_connection> | <
    operation_connection> | <property_set>
<instance_def>          <- 'instance' <identifier> ':' <identifier>
<port_connection>       <- 'connection' <identifier> '.' <identifier> '->' ('[' 
     <intvalue> ']')? <identifier> '.' <identifier>
<operation_connection> <- 'operation' <identifier> '.' <identifier> '->' <
    identifier> '.' <identifier>
<property_set>          <- 'property' <identifier> '.' <identifier> '=' <
    expression>
```

A.1.10 Définition de déploiements

A.1.10.1 Exemple

```
deployment Deployment {
  architecture Architecture

  property control.prop = 10

  activity control {
    priority = 8
    period   = 100
    deadline = 100
  }
  // ...
}
```

A.1.10.2 Syntaxe

```
<deployment_def>      <- 'deployment' <identifier> '{' <deployment_element>* '}'
<deployment_element> <- 'architecture' <architecture_ref> | <property_set> | <activity_def>
<activity_def>       <- 'activity' <identifier> '{' (<affinity_def>)? <priority_def> <period_def> (<deadline_def>)? '}'
<affinity_def>       <- 'affinity' '=' <intvalue>
<priority_def>       <- 'priority' '=' <intvalue> | 'background'
<period_def>         <- 'period' '=' <intvalue>
<deadline_def>       <- 'deadline' '=' <intvalue>
```

A.2 Langage d'expression

A.2.1 Programmes et instructions

A.2.1.1 Exemple

```
var b: float = testVarInit + 1.0;
if (a > 0) then
  testVar = a + b;
else
  testVar = b - a;
if (read(testInput, testVar) == new_data) then {
  testVarInit = 2.0;
  testVarInit = 2.0e10;
}
else {
  testVarInit = 01.0;
  testVarInit = 0.0;
```

```
13 }
14 write(testOutput, testVarInit);
15 return testVar;
```

A.2.1.2 Syntaxe

```
1 <program>    <- '{' <variable_def>* <statement>* '}'
2 <statement>  <- <expression> | <return> | <ite> | <block> | <assign>
3 <return>     <- 'return' <expression> ';'
4 <ite>        <- 'if' <expression> 'then' <statement> 'else' <statement>
5 <block>      <- '{' <statement>* '}'
6 <assign>     <- <expression> '=' <expression> ';'
```

A.2.2 Expressions

A.2.2.1 Syntaxe

```
1 <expression>         <- <boolvalue> | <intvalue> | <floatvalue> | <stringvalue>
      |
2                         <flowstatus> | <callstatus> | <activity_elements> |
3                         <variable_ref> | <codelcall> | <read> | <write> | <send
                              > | <collect> |
4                         <parentheses> | <unary_ops> | <binary_ops>
5 <boolvalue>          <- 'true' | 'false'
6 <flowstatus>         <- 'no_data' | 'new_data' | 'old_data'
7 <callstatus>         <- 'success' | 'no_success'
8 <activity_elements>  <- 'affinity' | 'priority' | 'period' | 'deadline'
9 <variable_ref>       <- <identifier>
10 <codelcall>         <- <identifier> '(' <args>? ')'
11 <args>              <- <expression> (',' <args>)?
12 <read>              <- 'read' '(' <identifier> ',' <variable_ref> ')'
13 <write>             <- 'write' '(' <identifier> ',' <variable_ref> ')'
14 <send>              <- 'send' '(' <identifier> (',' <args>)? ')'
15 <collect>           <- 'collect' '(' <identifier> (',' <variable_ref>)? ')'
16 <parentheses>       <- '(' <expression> ')'
17 <unary_ops>         <- <uop> <expression>
18 <uop>               <- '-' | '!'
19 <binary_ops>        <- <expression> <bop> <expression>
20 <bop>               <- '*' | '/' | '+' | '-' | '&&' | '||' | '==' | '!=' | '>'
      | '>=' | '<' | '<='
```

Annexe B

Exemple de librairie

Cette annexe contient un exemple complet de librairie, à partir du modèle Mauve jusqu'au code C++ généré.

Listing B.1 – Entête générée pour un composant

```
#ifndef MY_LIB_FOO_COMPONENT_H_
#define MY_LIB_FOO_COMPONENT_H_
#include <rtt/RTT.hpp>
#include "my_lib/my_lib.hpp"
namespace my_lib {
class Foo : public RTT::TaskContext {
public: Foo(const std::string& name);
private: int period;
private: int priority;
private: int deadline;
private: int affinity;
protected: virtual bool setPeriod(RTT::Seconds);
private: RTT::OutputPort<std::string> state_port;
protected: ::my_lib::A K;
private:
std::map<std::string, bool> hasNewData;
std::map<std::string, bool> hasBeenRead;
void new_data_callback(RTT::base::PortInterface* p) {
hasNewData[p->getName()] = true;
};
void resetPorts() {
std::map<std::string, bool>::iterator it;
for (it = hasNewData.begin(); it != hasNewData.end(); ++it)
it->second = false;
for (it = hasBeenRead.begin(); it != hasBeenRead.end(); ++it)
it->second = false;
};
template <typename T> RTT::FlowStatus read(RTT::InputPort<T> &p, T &d) {
hasBeenRead[p.getName()] = true;
return p.read(d);
};
protected: RTT::InputPort< int > input_port;
protected: RTT::OutputPort< int > output_port;
private: int input_value;
private: int output_value;
private: ::my_lib::B x;
protected: virtual bool configureHook();
protected: virtual void stopHook();
protected: virtual void cleanupHook();
protected: virtual bool startHook();
```

```
41 protected: virtual bool userStartHook();
42 protected: virtual void updateHook();
43 protected: virtual void userUpdateHook();
44 }; // component Foo
45 } // namespace my_lib
46 #endif // MY_LIB_FOO_COMPONENT_H_
```

Listing B.2 – Code source généré pour un composant

```
1
2 #include <rtt/Component.hpp>
3 #include "Foo-component.hpp"
4 #ifdef USE_LTTNG
5 #  define TRACEPOINT_DEFINE
6 #  define TRACEPOINT_PROBE_DYNAMIC_LINKAGE
7 #  include <my_lib-mauve-tp.h>
8 #  include <unistd.h>
9 #  include <sys/syscall.h>
10 #  include <sys/types.h>
11 #endif
12 namespace my_lib {
13 Foo::Foo ( const std::string& name )
14 : RTT::TaskContext(name, PreOperational)
15 , period(0), priority(0), affinity(0), deadline(0)
16 {
17 this->addProperty("K", this->K);
18 this->addEventPort("input_port", this->input_port, boost::bind(&Foo::
       new_data_callback, this, _1));
19 hasBeenRead[input_port.getName()] = false;
20 hasNewData[input_port.getName()] = false;
21 this->addPort("output_port", this->output_port);
22
23 this->addAttribute("period", period);
24 this->addAttribute("priority", priority);
25 this->addAttribute("deadline", deadline);
26 this->addAttribute("affinity", affinity);
27 this->addPort("state", state_port);
28 }
29 bool Foo::configureHook() {
30 return true;
31 }
32 bool Foo::userStartHook() {
33 return true;
34 }
35 bool Foo::startHook() {
36 bool r = userStartHook();
37
38 // init variables for PtLTL monitors
39
40 // write values in PtLTL ports
41
42 return r;
43 }
44 void Foo::cleanupHook() {
45 }
46 void Foo::stopHook() {
47 }
48 void Foo::userUpdateHook() {
49 {
```

```
50 {
51 rread< int >(input_port,this->input_value);
52 this->output_value = (this->input_value + 1);
53 output_port.write(this->output_value);
54 };
55 }}
56 void Foo::updateHook() {
57 #ifdef USE_LTTNG
58 tracepoint(mauve_my_lib, component, syscall(SYS_gettid), (char*)"begin", (char
       *)this->getName().c_str());
59 #endif
60
61 // PtLTL monitors checked before the execution of the component
62
63 // execution of component
64 userUpdateHook();
65 // PtLTL monitors checked after the execution of the component
66
67 // update ptltl variables for monitors
68
69 // write values in PtLTL monitors
70
71 resetPorts();
72 #ifdef USE_LTTNG
73 tracepoint(mauve_my_lib, component, syscall(SYS_gettid), (char*)"end", (char*)
       this->getName().c_str());
74 #endif
75
76 }
77
78 bool Foo::setPeriod(RTT::Seconds s) {
79 bool r = RTT::TaskContext::setPeriod(s);
80 if (r) period = s * 1000;
81 return r;
82 }
83 } // namespace my_lib
84 // Orocos Component
85 ORO_CREATE_COMPONENT(my_lib::Foo)
```

Annexe C

Modèles Mauve de l'architecture robotique d'exploration

C.1 Pilotes

C.1.1 Télémètre laser

C.1.1.1 Codels

```
1  library libhokuyo {
2      use sensor_msgs
3      use ctypes
4      use std
5
6      type LaserPtr
7      type LaserScanPtr
8
9      codel get_scan_from_ptr(LaserScanPtr): LaserScan
10     codel newLaserScanPtr(): LaserScanPtr
11     codel newLaserPtr(): LaserPtr
12
13     codel open_laser(string, LaserPtr): bool
14     codel config_scan(float, LaserPtr, LaserScanPtr, float): void
15     codel laser_on(LaserPtr): bool
16     codel get_scan(LaserPtr, LaserScanPtr): void
17     codel laser_off(LaserPtr): void
18     codel close_laser(LaserPtr): void
19 }
```

C.1.1.2 Composant

```
1  library laser {
2
3      use sensor_msgs
4      use libhokuyo
5      use std
6      use ctypes
7
```

```
8       shell LaserShell {
9           property span: float = 180.0
10          property maxRange: float = 10.0
11          property device: string
12          output port scan_port: LaserScan
13      }
14
15      core HokuyoCore(LaserShell) {
16          var laser: LaserPtr
17          var scan: LaserScanPtr
18
19          configure {
20              laser = newLaserPtr();
21              if (open_laser(device, laser)) then {
22                  scan = newLaserScanPtr();
23                  config_scan(span, laser, scan, maxRange);
24              } else {
25                  return false;
26              }
27              return true;
28          }
29
30          start {
31              return laser_on(laser);
32          }
33
34          update {
35              get_scan(laser, scan);
36              write(scan_port, get_scan_from_ptr(scan));
37          }
38
39          stop {
40              laser_off(laser);
41          }
42
43          cleanup {
44              close_laser(laser);
45          }
46      }
47
48      component Hokuyo(LaserShell, HokuyoCore)
49
50      architecture HokuyoTest {
51          instance hokuyo: Hokuyo
52      }
53  }
```

C.1.2 Robot

C.1.2.1 Codels

```
1  library libaria {
2      use geometry_msgs
3      use sensor_msgs
4      use ctypes
```

```
5     use std
6
7     type ArRobotPtr
8     type ArArgumentBuilderPtr
9     type ArArgumentParserPtr
10    type ArRobotConnectorPtr
11    type ArSonarDevicePtr
12    type ArPosePtr
13
14    codel newArRobotConnectorPtr(ArArgumentParserPtr, ArRobotPtr):
          ArRobotConnectorPtr
15    codel newArPosePtr(): ArPosePtr
16    codel newArSonarDevicePtr(ArRobotPtr): ArSonarDevicePtr
17    codel newArArgumentParserPtr(ArArgumentBuilderPtr, string):
          ArArgumentParserPtr
18    codel newArArgumentBuilder(): ArArgumentBuilderPtr
19    codel newArRobotPtr(): ArRobotPtr
20    codel aria_init(): void
21    codel aria_start(ArRobotPtr, ArArgumentParserPtr, ArRobotConnectorPtr,
          bool, bool): bool
22    codel aria_setWheels(ArRobotPtr, Twist): void
23    codel aria_getSonar(ArRobotPtr): LaserScan
24    codel aria_getPose(ArRobotPtr, ArPosePtr): PoseStamped
25    codel aria_getSpeed(ArRobotPtr): Twist
26    codel aria_stop(ArRobotPtr): void
27    codel aria_cleanup(ArRobotPtr, ArArgumentBuilderPtr, ArArgumentParserPtr,
          ArRobotConnectorPtr, ArSonarDevicePtr, ArPosePtr): void
28    codel aria_loop_once(ArRobotPtr): void
29 }
```

C.1.2.2 Composant

```
1  library p3dx {
2
3    use sensor_msgs
4    use geometry_msgs
5    use libaria
6    use std
7    use ctypes
8
9    shell P3DXShell {
10     input port actuator: Twist
11     output port sonar: LaserScan
12     output port pose: PoseStamped
13     output port speed: Twist
14     property sonarOnOff: bool
15     property motorOnOff: bool
16     property serial: string //"-rp /dev/ttyUSB0 -rb 9600"
17   }
18
19   core P3DXCore (P3DXShell) {
20     var actuator_sample: Twist
21     var sonar_sample: LaserScan
22     var pose_sample: PoseStamped
23     var speed_sample: Twist
24
```

```
25      var robot: ArRobotPtr
26      var builder: ArArgumentBuilderPtr
27      var parser: ArArgumentParserPtr
28      var robotConnector: ArRobotConnectorPtr
29      var sonarDev: ArSonarDevicePtr
30      var pose_var: ArPosePtr
31
32      configure {
33        builder = newArArgumentBuilder();
34        parser = newArArgumentParserPtr(builder, serial);
35        robot = newArRobotPtr();
36        robotConnector = newArRobotConnectorPtr(parser, robot);
37        sonarDev = newArSonarDevicePtr(robot);
38        pose_var = newArPosePtr();
39        aria_init();
40        return true;
41      }
42
43      start {
44        return aria_start(robot, parser, robotConnector, motorOnOff, sonarOnOff)
              ;
45      }
46
47      update {
48        read(actuator, actuator_sample);
49        aria_setWheels(robot, actuator_sample);
50        aria_loop_once(robot);
51        sonar_sample = aria_getSonar(robot);
52        speed_sample = aria_getSpeed(robot);
53        pose_sample = aria_getPose(robot, pose_var);
54        write(sonar, sonar_sample);
55        write(speed, speed_sample);
56        write(pose, pose_sample);
57      }
58
59      stop {
60        aria_stop(robot);
61      }
62
63      cleanup {
64        aria_cleanup(robot, builder, parser, robotConnector, sonarDev, pose_var)
              ;
65      }
66    }
67
68    component P3DX (P3DXShell, P3DXCore)
69
70    architecture P3DXTest {
71      instance p3dx: P3DX
72    }
73  }
```

C.2 Localisation

C.2.1 Localisation et cartographie simultanée (SLAM)

```
1  library gmapping {
2
3      use geometry_msgs
4      use sensor_msgs
5      use nav_msgs
6      use std
7      use ctypes
8      use libfastslam
9
10     type RangeSensorPtr
11     codel newRangeSensorPtr(): RangeSensorPtr
12
13     type GridSlamProcessorPtr
14     codel newGridSlamProcessorPtr(): GridSlamProcessorPtr
15
16     type MapSizePtr
17     codel newMapSizePtr(double, double, double, double): MapSizePtr
18
19     codel initMapper(LaserScan, GridSlamProcessorPtr, RangeSensorPtr,
           PoseStamped, double, double, MapSizePtr): RangeSensorPtr
20     codel addScan(LaserScan, PoseStamped, GridSlamProcessorPtr, RangeSensorPtr
           ): bool
21     codel poseDifference(PoseStamped, PoseStamped): PoseStamped
22     codel bestParticlePose(GridSlamProcessorPtr): PoseStamped
23     codel updateMap(GridSlamProcessorPtr, RangeSensorPtr, LaserScan,
           OccupancyGridPtr, double, double, MapSizePtr): void
24
25     shell SLAMShell {
26         input port scan_port: LaserScan
27         input port input_pose: PoseStamped
28         output port pose_correction: PoseStamped
29         output port output_pose: PoseStamped
30         output port map_port: OccupancyGrid
31         output port entropy: double
32         property resolution: double
33         property size: double
34         property pgm_file: string
35         property particles: int = 30
36         property maxUrange: double = 10.0
37         property maxRange: double = 10.0
38         property xmin: double = -5.0
39         property xmax: double = 5.0
40         property ymin: double = -5.0
41         property ymax: double = 5.0
42     }
43
44     core GMappingCore (SLAMShell) {
45       var gsp: GridSlamProcessorPtr
46         var gsp_laser: RangeSensorPtr
47
48         var scan_sample: LaserScan
49         var pose_sample: PoseStamped
50         var estimated_pose: PoseStamped
```

```
51          var pose_difference: PoseStamped
52
53          var map: OccupancyGridPtr
54          var map_size: MapSizePtr
55
56          start {
57              gsp = newGridSlamProcessorPtr();
58              gsp_laser = newRangeSensorPtr();
59              map = newOccupancyGridPtr();
60              map_size = newMapSizePtr(xmin, xmax, ymin, ymax);
61          }
62
63          update StateMachine GMappingStateMachine
64      }
65
66      StateMachine GMappingStateMachine {
67          var scan_received: bool;
68          var scan_processed: bool;
69          var count_processing: int;
70
71          initial state Uninitialized {
72              run {
73                  read(input_pose, pose_sample);
74                  scan_received = (read(scan_port, scan_sample) == NewData);
75              }
76              exit {
77                  gsp_laser = initMapper(scan_sample, gsp, gsp_laser,
                        pose_sample, maxRange, maxUrange, map_size);
78              }
79              transition if (scan_received) select ProcessingScan
80          }
81
82          state ProcessingScan {
83              entry {
84                  count_processing = 1;
85              }
86              run {
87                  read(scan_port, scan_sample);
88                  read(input_pose, pose_sample);
89                  scan_processed = addScan(scan_sample, pose_sample, gsp,
                        gsp_laser);
90                  estimated_pose = bestParticlePose(gsp);
91                  write(output_pose, estimated_pose);
92                  pose_difference = poseDifference(pose_sample, estimated_pose);
93                  write(pose_correction, pose_difference);
94                  count_processing = count_processing + 1;
95              }
96              transition if ((count_processing > 0) && scan_processed)
97                  select UpdatingMap
98          }
99
100         state UpdatingMap {
101             entry {
102                 updateMap(gsp, gsp_laser, scan_sample, map, maxRange,
                        maxUrange, map_size);
103                 write(map_port, getMap(map));
104             }
105             transition select ProcessingScan
106         }
```

```
107     }
108
109     component GMapping (SLAMShell, GMappingCore)
110
111     architecture SLAMTest {
112         instance gmapping: GMapping
113     }
114 }
```

C.2.2 Maintien de la pose

```
1  library posecorrection {
2      use ctypes
3      use nav_msgs
4      use geometry_msgs
5      use gridplanning
6      use libcontrol
7
8      codel pose_correction(PoseStamped, PoseStamped): PoseStamped
9
10     shell PoseCorrectionShell {
11         input port odom_pose: PoseStamped
12         input port correction: PoseStamped
13         output port corrected_pose: PoseStamped
14     }
15
16     core PoseCorrectionCore (PoseCorrectionShell) {
17         var odom_sample: PoseStamped
18         var correction_sample: PoseStamped
19         var pose_sample: PoseStamped
20         update StateMachine PoseCorrectionStateMachine
21     }
22     StateMachine PoseCorrectionStateMachine {
23         var one_pose: bool = false;
24
25         initial state Idle {
26             run {
27                 one_pose = (read(correction, correction_sample) == NewData);
28             }
29             transition if (one_pose) select PoseCorrection
30         }
31         state PoseCorrection {
32             run {
33                 read(odom_pose, odom_sample);
34                 read(correction, correction_sample);
35                 pose_sample = pose_correction(odom_sample, correction_sample);
36                 write(corrected_pose, pose_sample);
37             }
38         }
39     }
40
41     component PoseCorrection (PoseCorrectionShell, PoseCorrectionCore)
42
43     architecture NavigationTest {
44         instance wavefront: WavefrontNavigation
45         instance astar: AstarNavigation
```

```
46          instance poc: PoseCorrection
47      }
48  }
```

C.3 Architecture de type navigation-guidage-contrôle

C.3.1 Exploration

```
1  library exploration {
2
3      use ctypes
4      use nav_msgs
5      use geometry_msgs
6
7      codel frontier(PoseStamped, OccupancyGrid): PoseStamped
8      codel equal_pose(PoseStamped, PoseStamped): bool
9
10     shell ExplorationShell {
11         input port map: OccupancyGrid
12         input port pose: PoseStamped
13         output port new_point: PoseStamped
14     }
15
16     core FrontierExplorationCore ( ExplorationShell ) {
17         var map_: OccupancyGrid
18         var pose_: PoseStamped
19         var target: PoseStamped
20
21         update StateMachine FrontierStateMachine
22     }
23
24     StateMachine FrontierStateMachine {
25         var has_map: bool;
26         var has_pose: bool;
27         var length: double;
28
29         initial state Stopped {
30             run {
31                 has_map = (read(map, map_) != NoData);
32                 has_pose = (read(pose, pose_) != NoData);
33             }
34             transition if (has_map && has_pose) select Exploring
35         }
36         state Exploring {
37             run {
38                 read(map, map_);
39                 read(pose, pose_);
40                 target = frontier(pose_, map_);
41             }
42             handle {
43                 /*send(h_cost, pose_, target);
44                 if (collect(h_cost, length) == Success) then {
45                     if (length >= 0) then {
46                         write(new_point, target);
```

```
47                      } else {}
48                  } else {}*/
49                  write(new_point, target);
50              }
51              transition if (equal_pose(target, pose_)) select Explored
52          }
53          state Explored {}
54      }
55
56      component FrontierExploration (ExplorationShell, FrontierExplorationCore)
57
58      architecture ExplorationTest {
59          instance frontier_exploration: FrontierExploration
60      }
61
62  }
```

C.3.2 Navigation

C.3.2.1 Codels

```
1  library gridplanning {
2
3      use ctypes
4      use nav_msgs
5      use geometry_msgs
6
7      codel wavefront(PoseStamped /* start */, PoseStamped /* goal */,
8          OccupancyGrid /* map */, double /* resolution */): Path
9
10     codel astar(PoseStamped /* start */, PoseStamped /* goal */,
11         OccupancyGrid /* map */, double /* resolution */): Path
12
13
14 }
```

C.3.2.2 Composant

```
1  library navigation {
2      use ctypes
3      use nav_msgs
4      use geometry_msgs
5      use gridplanning
6      use libcontrol
7
8      codel cost_of_path(PoseStamped, Path, OccupancyGrid): double
9      codel getPose(Path, int): PoseStamped
10     codel getLength(Path): int
11
12     shell NavigationShell {
13         property resolution: double = 0.3
14         input port map: OccupancyGrid
```

```
15          input port pose: PoseStamped
16          input port goal: PoseStamped
17          output port path: Path
18          output port next_point: PoseStamped
19          provide cost(PoseStamped /* start */, PoseStamped /* goal */): double
                /* length */
20      }
21
22      core WavefrontCore (NavigationShell) {
23          var pose_: PoseStamped
24          var goal_: PoseStamped
25          var map_: OccupancyGrid
26          var path_: Path
27          var new_goal: bool = false
28          var current_goal: PoseStamped
29
30          operation cost(s, g) = {
31              path_ = wavefront(s, g, map_, resolution);
32              return cost_of_path(s, path_, map_);
33          }
34
35          update StateMachine WavefrontStateMachine
36      }
37
38      StateMachine WavefrontStateMachine {
39          var next_goal: int = 0;
40          var path_length: int = 0;
41
42          initial state Idle {
43              run {
44                  new_goal = (read(goal, goal_) == NewData);
45              }
46              transition if (new_goal) select Planning
47          }
48          state Planning {
49              entry {
50                  read(pose, pose_);
51                  read(map, map_);
52                  path_ = wavefront(pose_, goal_, map_, resolution);
53                  write(path, path_);
54                  next_goal = 0;
55                  path_length = getLength(path_);
56              }
57              transition if (path_length != 0) select Navigating
58              transition if (path_length == 0) select Idle
59          }
60          state Navigating {
61              entry {
62                  current_goal = getPose(path_, next_goal);
63                  next_goal = next_goal + 1;
64                  write(next_point, current_goal);
65              }
66              run {
67                  new_goal = (read(goal, goal_) == NewData);
68                  read(pose, pose_);
69                  read(map, map_);
70              }
71              transition if (new_goal) select Planning
72              transition if (is_arrived(pose_, current_goal, 0.5)
```

```
73                  && (next_goal == path_length)) select Idle
74              transition if (is_arrived(pose_, current_goal, 0.5)) select
                    Navigating
75          }
76      }
77
78      component WavefrontNavigation (NavigationShell, WavefrontCore)
79
80      core AstarCore (NavigationShell) {
81          var pose_: PoseStamped
82          var goal_: PoseStamped
83          var map_: OccupancyGrid
84          var path_: Path
85          var new_goal: bool = false
86          var current_goal: PoseStamped
87
88          operation cost(s, g) = {
89              path_ = astar(s, g, map_, resolution);
90              return cost_of_path(s, path_, map_);
91          }
92
93          update StateMachine AstarStateMachine
94      }
95
96      StateMachine AstarStateMachine {
97          var next_goal: int = 0;
98          var path_length: int = 0;
99
100         initial state Idle {
101             run {
102                 new_goal = (read(goal, goal_) == NewData);
103             }
104             transition if (new_goal) select Planning
105         }
106         state Planning {
107             entry {
108                 read(pose, pose_);
109                 read(map, map_);
110                 path_ = astar(pose_, goal_, map_, resolution);
111                 write(path, path_);
112                 next_goal = 0;
113                 path_length = getLength(path_);
114             }
115             transition select Navigating
116         }
117         state Navigating {
118             entry {
119                 current_goal = getPose(path_, next_goal);
120                 next_goal = next_goal + 1;
121                 write(next_point, current_goal);
122             }
123             run {
124                 new_goal = (read(goal, goal_) == NewData);
125                 read(pose, pose_);
126                 read(map, map_);
127             }
128             transition if (new_goal) select Planning
129             transition if (is_arrived(pose_, current_goal, 0.5)
130                 && (next_goal == path_length)) select Idle
```

```
131             transition if (is_arrived(pose_, current_goal, 0.5)) select
                    Navigating
132         }
133     }
134
135     component AstarNavigation (NavigationShell, AstarCore)
136
137     architecture NavigationTest {
138         instance wavefront: WavefrontNavigation
139         instance astar: AstarNavigation
140         instance poc: PoseCorrection
141     }
142 }
```

C.3.3 Guidage

```
1  library guidance {
2
3      use ctypes
4      use geometry_msgs
5      use sensor_msgs
6      use libcontrol
7
8      shell GuidanceShell {
9          property position_tolerance: double = 0.2
10         property obstacle_distance: double = 1.0
11         input port pose: PoseStamped
12         input port goal: PoseStamped
13         input port scan: LaserScan
14         output port setpoint: PoseStamped
15     }
16
17     core ObstacleAvoidanceCore (GuidanceShell) {
18         var pose_: PoseStamped
19         var goal_: PoseStamped
20         var scan_: LaserScan
21         var target: PoseStamped
22
23         stop {
24             read(pose, pose_);
25             write(setpoint, pose_);
26         }
27         update StateMachine ObstacleStateMachine
28     }
29
30     StateMachine ObstacleStateMachine {
31         var new_goal: bool;
32
33         initial state Stopped {
34             entry {
35                 read(pose, pose_);
36                 write(setpoint, pose_);
37             }
38             run {
39                 new_goal = (read(goal, goal_) != NoData);
40             }
```

```
41              transition if (new_goal) select Moving
42          }
43          state Moving {
44              run {
45                  read(pose, pose_);
46                  read(goal, goal_);
47                  read(scan, scan_);
48              }
49              handle {
50                  target = nextPoint(pose_, goal_, scan_, obstacle_distance);
51                  write(setpoint, target);
52              }
53              transition if (is_arrived(pose_, goal_, position_tolerance))
                    select Stopped
54          }
55      }
56
57      core VFHCore (GuidanceShell) {
58          var pose_: PoseStamped
59          var goal_: PoseStamped
60          var scan_: LaserScan
61          var target: PoseStamped
62          var sector_width: int = 10
63
64          start {
65              return (read(pose, pose_) != NoData);
66          }
67          stop {
68              read(pose, pose_);
69              write(setpoint, pose_);
70          }
71          update StateMachine VFHStateMachine
72      }
73
74      StateMachine VFHStateMachine {
75          var new_goal: bool;
76
77          initial state Stopped {
78              entry {
79                  read(pose, pose_);
80                  write(setpoint, pose_);
81              }
82              run {
83                  new_goal = (read(goal, goal_) != NoData);
84              }
85              transition if (new_goal) select Moving
86          }
87          state Moving {
88              run {
89                  read(pose, pose_);
90                  read(goal, goal_);
91                  read(scan, scan_);
92              }
93              handle {
94                  target = vfh(pose_, goal_, scan_, sector_width,
                        obstacle_distance);
95                  write(setpoint, target);
96              }
```

```
97              transition if (is_arrived(pose_, goal_, position_tolerance))
                    select Stopped
98          }
99      }
100
101     core FuzzyGuidanceCore (GuidanceShell) {
102         var pose_: PoseStamped
103         var goal_: PoseStamped
104         var scan_: LaserScan
105         var target: PoseStamped
106
107         start {
108             return (read(pose, pose_) != NoData);
109         }
110         stop {
111             read(pose, pose_);
112             write(setpoint, pose_);
113         }
114         update StateMachine FuzzyGuidanceStateMachine
115     }
116
117     StateMachine FuzzyGuidanceStateMachine {
118         var new_goal: bool;
119
120         initial state Stopped {
121             entry {
122                 read(pose, pose_);
123                 write(setpoint, pose_);
124             }
125             run {
126                 new_goal = (read(goal, goal_) != NoData);
127             }
128             transition if (new_goal) select Moving
129         }
130         state Moving {
131             run {
132                 read(pose, pose_);
133                 read(goal, goal_);
134                 read(scan, scan_);
135             }
136             handle {
137                 target = fuzzy_obstacle_avoidance(pose_, goal_, scan_,
                        obstacle_distance);
138                 write(setpoint, target);
139             }
140             transition if (is_arrived(pose_, goal_, position_tolerance))
                    select Stopped
141         }
142     }
143
144     component VFH (GuidanceShell, VFHCore)
145     component FuzzyGuidance (GuidanceShell, FuzzyGuidanceCore)
146     component ObstacleAvoidance (GuidanceShell, ObstacleAvoidanceCore)
147
148     architecture GuidanceTest {
149         instance obstacle_avoidance: ObstacleAvoidance
150         instance fuzzy_guidance: FuzzyGuidance
151         instance vfh: VFH
152     }
```

```
153
154 }
```

C.3.4 Contrôle

C.3.4.1 Codels

```
1  library libcontrol {
2      use ctypes
3      use geometry_msgs
4      use sensor_msgs
5
6      type double_ptr
7      codel init_double_ptr(): double_ptr
8
9      codel stop_control(): Twist
10     codel is_arrived(PoseStamped, PoseStamped, double): bool
11
12     // Unicycle
13     codel unicycle_steering(PoseStamped, PoseStamped, double, double): Twist
14
15     // PID
16     codel pid_control(PoseStamped, PoseStamped, double, double, double, double
           ,
17     double_ptr, double_ptr, double_ptr, double, double): Twist
18
19     // Fuzzy controller
20     codel fuzzy_obstacle_avoidance(PoseStamped, PoseStamped, LaserScan, double
           ): PoseStamped
21
22     // Virtual Force Field
23     codel vfh(PoseStamped, PoseStamped, LaserScan, int, double): PoseStamped
24
25     // ObstacleAvoidance
26     codel nextPoint(PoseStamped, PoseStamped, LaserScan, double): PoseStamped
27 }
```

C.3.4.2 Composant

```
1  library control {
2
3      use ctypes
4      use geometry_msgs
5      use sensor_msgs
6      use libcontrol
7
8      shell ControlShell {
9          property position_tolerance: double = 0.2
10         property orientation_tolerance: double = 0.05
11
12         property Pgain: double = 5.0
13         property Igain: double = 0.0
```

```
14          property Dgain: double = 1.0
15
16          property v_max: double = 1.0
17          property w_max: double = 0.2
18
19          input port pose: PoseStamped
20          input port setpoint: PoseStamped
21          output port control: Twist
22      }
23
24      core UnicycleSteeringCore (ControlShell) {
25          var current_pose: PoseStamped
26          var goal_pose: PoseStamped
27          var speed: Twist
28
29          update StateMachine UnicycleStateMachine
30          stop { write(control, stop_control()); }
31      }
32
33      StateMachine UnicycleStateMachine {
34          var new_goal: bool;
35          var new_pose: bool;
36
37          initial state Stopped {
38             entry {
39                 write(control, stop_control());
40             }
41             run {
42                 new_goal = (read(setpoint, goal_pose) != NoData);
43                 new_pose = (read(pose, current_pose) != NoData);
44             }
45             transition if (new_goal && new_pose) select Moving
46          }
47          state Moving {
48              run {
49                  read(pose, current_pose);
50                  read(setpoint, goal_pose);
51              }
52              handle {
53                  speed = unicycle_steering(current_pose, goal_pose,
                        position_tolerance, orientation_tolerance);
54                  write(control, speed);
55              }
56              transition if (is_arrived(current_pose, goal_pose,
                    position_tolerance)) select Arrived
57          }
58          state Arrived {
59              entry {
60                  write(control, stop_control());
61              }
62              run {
63                  new_goal = (read(setpoint, goal_pose) == NewData);
64              }
65              transition if (new_goal) select Moving
66          }
67      }
68
69      component UnicycleSteering (ControlShell, UnicycleSteeringCore)
70
```

```
71      core PIDHeadingCore (ControlShell) {
72          var current_pose: PoseStamped
73          var goal_pose: PoseStamped
74          var speed: Twist
75          var e1: double_ptr
76          var e2: double_ptr
77          var k1: double_ptr
78
79          configure {
80              return true;
81          }
82          start {
83              k1 = init_double_ptr();
84              e1 = init_double_ptr();
85              e2 = init_double_ptr();
86              return true;
87          }
88          update StateMachine PIDHeadingStateMachine
89          stop { write(control, stop_control()); }
90      }
91
92      StateMachine PIDHeadingStateMachine {
93          var new_goal: bool;
94          var new_pose: bool;
95
96          initial state Stopped {
97              entry {
98                  write(control, stop_control());
99              }
100             run {
101                 new_goal = (read(setpoint, goal_pose) != NoData);
102                 new_pose = (read(pose, current_pose) != NoData);
103             }
104             transition if (new_goal && new_pose) select Moving
105         }
106         state Moving {
107             run {
108                 read(pose, current_pose);
109                 read(setpoint, goal_pose);
110             }
111             handle {
112                 speed = pid_control(current_pose, goal_pose,
113                     Pgain, Igain, Dgain, (period/1000.0), k1, e1, e2, v_max,
                        w_max);
114                 write(control, speed);
115             }
116             transition if (is_arrived(current_pose, goal_pose,
                    position_tolerance)) select Arrived
117         }
118         state Arrived {
119             entry {
120                 write(control, stop_control());
121             }
122             run {
123                 new_goal = (read(setpoint, goal_pose) == NewData);
124             }
125             transition if (new_goal) select Moving
126         }
127     }
```

```
128
129     component PIDControl (ControlShell, PIDHeadingCore)
130
131     architecture ControlTest {
132         instance steering: UnicycleSteering
133         instance pid_control: PIDControl
134     }
135
136 }
```

C.3.5 Sécurité

C.3.5.1 Codels

```
1 library libswitch {
2     use ctypes
3     use geometry_msgs
4     use sensor_msgs
5
6     codel hitPrevention(LaserScan, Twist, double, double, double): Twist
7 }
```

C.3.5.2 Composant

```
1 library safetyswitch {
2
3     use ctypes
4     use std
5     use geometry_msgs
6     use std_msgs
7     use sensor_msgs
8     use libswitch
9
10    codel modeSwitch(String): int
11
12    shell SafetySwitchShell {
13        property distance_stop: double = 0.25
14        property distance_slow: double = 0.5
15        property robotWidth: double = 0.26
16
17      input port manualInput: Twist
18        input port autonomousInput: Twist
19        input port scanInput: LaserScan
20        output port outputTwist: Twist
21        input port mode: String
22    }
23
24    core SafetySwitchCore (SafetySwitchShell) {
25        var transitionMode: int = 2
26        var transitionString: String
27        var transitionNew: bool = false
28        var manual_twist: Twist
```

```
29          var autonomous_twist: Twist
30          var scan: LaserScan
31
32          update StateMachine SafetySwitchStateMachine
33      }
34
35      StateMachine SafetySwitchStateMachine {
36
37          initial state ManualAssisted {
38              run {
39                  read(manualInput, manual_twist);
40                  read(scanInput, scan);
41                  transitionNew = (read(mode, transitionString) == NewData);
42                  transitionMode = modeSwitch(transitionString);
43              }
44              handle {
45                  write(outputTwist, hitPrevention(scan, manual_twist,
                        distance_slow, distance_stop, robotWidth));
46              }
47              transition if (transitionNew && transitionMode == 1) select Manual
48              transition if (transitionNew && transitionMode == 3) select
                    Autonomous
49              transition if (transitionNew && transitionMode == 4) select
                    AutonomousAssisted
50          }
51          state Manual {
52        run {
53          read(manualInput, manual_twist);
54                  transitionNew = (read(mode, transitionString) == NewData);
55          transitionMode = modeSwitch(transitionString);
56        }
57        handle {
58          write(outputTwist, manual_twist);
59        }
60        transition if (transitionNew && transitionMode == 2) select
              ManualAssisted
61        transition if (transitionNew && transitionMode == 3) select Autonomous
62        transition if (transitionNew && transitionMode == 4) select
              AutonomousAssisted
63      }
64          state Autonomous {
65              run {
66                  read(autonomousInput, autonomous_twist);
67                  transitionNew = (read(mode, transitionString) == NewData);
68                  transitionMode = modeSwitch(transitionString);
69              }
70              handle {
71                  write(outputTwist, autonomous_twist);
72              }
73              transition if (transitionNew && transitionMode == 1) select Manual
74              transition if (transitionNew && transitionMode == 2) select
                    ManualAssisted
75              transition if (transitionNew && transitionMode == 4) select
                    AutonomousAssisted
76          }
77          state AutonomousAssisted {
78              run {
79                  read(autonomousInput, autonomous_twist);
80                  read(scanInput, scan);
```

```
81                  transitionNew = (read(mode, transitionString) == NewData);
82                  transitionMode = modeSwitch(transitionString);
83              }
84              handle {
85                  write(outputTwist, hitPrevention(scan, autonomous_twist,
                        distance_slow, distance_stop, robotWidth));
86              }
87              transition if (transitionNew && transitionMode == 1) select Manual
88              transition if (transitionNew && transitionMode == 2) select
                    ManualAssisted
89              transition if (transitionNew && transitionMode == 3) select
                    Autonomous
90          }
91      }
92
93      component SafetySwitch (SafetySwitchShell, SafetySwitchCore)
94
95      architecture SafetySwitchTest {
96          instance safetySwitch: SafetySwitch
97      }
98
99  }
```

Annexe D

Pires temps d'exécution des codels et des composants de l'architecture robotique d'exploration

Cette annexe présente les pires temps d'exécution probabilistes (WCET) des différents composants, de la glue de ces composants ainsi que des codels.

D.1 Pilotes

D.1.1 Télémètre laser

Élément	Seuil de probabilité	WCET (ns)
composant	$1 \cdot 10^{-07}$	31 621 229,834 170 9
glue composant	$1 \cdot 10^{-07}$	240 089,371 859 296
get_scan	$1 \cdot 10^{-07}$	36 689 965,753 768 8
get_scan_from_ptr	$1 \cdot 10^{-07}$	59 879,889 447 236 2

D.1.2 Robot

Élément	Seuil de probabilité	WCET (ns)
composant	$1 \cdot 10^{-07}$	17 811 115,165 829 1
glue composant	$1 \cdot 10^{-07}$	448 564,336 683 417
aria_setWheels	$1 \cdot 10^{-07}$	243 150,648 241 206
aria_loop_once	$1 \cdot 10^{-07}$	17 920 071,633 165 8
aria_getSonar	$1 \cdot 10^{-07}$	103 864,371 859 296
aria_getSpeed	$1 \cdot 10^{-07}$	44 044,824 120 603
aria_getPose	$1 \cdot 10^{-07}$	183 764,316 582 915

D.2 Localisation

D.2.1 Localisation et cartographie simultanée (SLAM)

Élément	Seuil de probabilité	WCET (ns)
composant	$1 \cdot 10^{-07}$	1 164 778 454,894 47
glue composant	$1 \cdot 10^{-07}$	537 716,909 547 739
addScan	$1 \cdot 10^{-07}$	1 378 923 279,080 4
bestParticlePose	$1 \cdot 10^{-07}$	45 307,457 286 432 2
poseDifference	$1 \cdot 10^{-07}$	52 239,713 567 839 2
updateMap	$1 \cdot 10^{-07}$	172 415 648,216 08
getMap	$1 \cdot 10^{-07}$	177 085,819 095 477

D.2.2 Maintien de la pose

Élément	Seuil de probabilité	WCET (ns)
composant	$1 \cdot 10^{-07}$	499 765,140 703 517
glue composant	$1 \cdot 10^{-07}$	352 312,623 115 578
pose_correction	$1 \cdot 10^{-07}$	387 060,909 547 739

D.3 Architecture de type navigation-guidage-contrôle

D.3.1 Exploration

Élément	Seuil de probabilité	WCET (ns)
composant	$1 \cdot 10^{-07}$	40 302 848,075 376 9
glue composant	$1 \cdot 10^{-07}$	1 609 031,658 291 46
frontier	$1 \cdot 10^{-07}$	38 062 486,859 296 5
equal_pose	$1 \cdot 10^{-07}$	26 464,678 391 959 8

D.3.2 Navigation

Élément	Seuil de probabilité	WCET (ns)
composant	$1 \cdot 10^{-07}$	5 287 817 189,180 9
glue composant	$1 \cdot 10^{-07}$	1 452 640,809 045 23
wavefront	$1 \cdot 10^{-07}$	3 816 335 207,115 58
getLength	$1 \cdot 10^{-07}$	22 748,256 281 407
getPose	$1 \cdot 10^{-07}$	104 821,804 020 101

D.3.3 Guidage

Élément	Seuil de probabilité	WCET (ns)
composant	$1 \cdot 10^{-07}$	840 449,120 603 016
glue composant	$1 \cdot 10^{-07}$	423 969,261 306 533
nextPoint	$1 \cdot 10^{-07}$	769 717,552 763 819

D.3.4 Contrôle

Élément	Seuil de probabilité	WCET (ns)
composant	$1 \cdot 10^{-07}$	1 947 007,804 020 1
glue composant	$1 \cdot 10^{-07}$	307 249,713 567 839
is_arrived	$1 \cdot 10^{-07}$	180 627,768 844 221
stop_control	$1 \cdot 10^{-07}$	15 057,587 939 698 5
pid_control	$1 \cdot 10^{-07}$	1 897 212,482 412 06

D.3.5 Sécurité

Élément	Seuil de probabilité	WCET (ns)
composant	$1 \cdot 10^{-07}$	474 212,964 824 121
glue composant	$1 \cdot 10^{-07}$	302 711,929 648 241
modeSwitch	$1 \cdot 10^{-07}$	95 728,246 231 155 8
hitPrevention	$1 \cdot 10^{-07}$	294 669,231 155 779

Bibliographie

Abdellatif, Tesnim et al. (2012). « Rigorous design of robot software : A formal component-based approach ». In : *Robotics and Autonomous Systems* 60.12, p. 1563–1578 (cf. p. 19).

Albus, James S (2002). « 4D/RCS : a reference model architecture for intelligent unmanned ground vehicles ». In : *AeroSense 2002*, p. 1–8 (cf. p. 6, 7).

Alonso, Diego et al. (2010). « V3CMM : a 3-View Component Meta-Model for Model-Driven Robotic Software Development ». In : *Journal of Software Engineering for Robotics (JOSER)* 1, p. 3–17 (cf. p. 19).

Altmeyer, Sebastian, Nicolas Navet et Loïc Fejoz (2015). « Using CPAL to model and validate the timing behaviour of embedded systems ». In : *6th International Workshop on Analysis Tools and Methodologies for Embedded and Real-time Systems (WATERS)* (cf. p. 17).

Anagnostopoulos, Christos-Nikolaos, Theodoros Iliou et Ioannis Giannoukos (2015). « Features and classifiers for emotion recognition from speech : a survey from 2000 to 2011 ». In : *Artificial Intelligence Review* 43.2, p. 155–177 (cf. p. 9).

Ando, Noriaki et al. (2011). « Software Deployment Infrastructure for Component Based RT-Systems ». In : *Journal of Robotics and Mechatronics* 23.3 (cf. p. 13).

Arkin, Ronald C et Tucker Balch (1997). « AurA : Principles and Practice in Review ». In : *Journal of Experimental and Theoretical Artificial Intelligence (JTAI)* 9, p. 175–189 (cf. p. 10).

Audsley, Neil et al. (1993). « Applying New Scheduling Theory To Static Priority Preemptive Scheduling ». In : *Software Engineering Journal* September, p. 284–292 (cf. p. 86).

Baker, Theodore P (1991). « Stack-based scheduling of realtime processes ». In : *Real-Time Systems* 3.1, p. 67–99 (cf. p. 28).

Ballabriga, Clément et al. (2010). « Otawa : An open toolbox for adaptive wcet analysis ». In : *Software Technologies for Embedded and Ubiquitous Systems*. Waidhofen, Austria, p. 35–46 (cf. p. 55).

Baruah, Sanjoy (2003). « Dynamic- and static-priority scheduling of recurring real-time tasks ». In : *Real-Time Systems* 24, p. 93–128 (cf. p. 29).

Baruah, Sanjoy, Deji Chen et al. (1999). « Generalized multiframe tasks ». In : *Real-Time Systems* 17.1, p. 5–22 (cf. p. 29).

Baruah, Sanjoy, Louis E Rosier et Rodney R Howell (1990). « Algorithms and complexity concerning the preemptive scheduling of periodic, real-time tasks on one processor ». In : *Real-Time Systems* 2, p. 301–324 (cf. p. 75).

Basu, Ananda, Marius Bozga et Joseph Sifakis (2006). « Modeling Heterogeneous Real-time Systems in BIP ». In : *SEFM06, Pune, India*, p. 3–12 (cf. p. 19).

Basu, Ananda, Matthieu Gallien et al. (2008). « Incremental Component-Based Construction and Verification of a Robotic System ». In : *Proceedings of the 18th European Conference on Artificial Intelligence*. Patras, Greece, p. 631–635 (cf. p. 19).

Beeden, Alexandra et Joost de Bruin (2010). « The Office ». In : *Television & New Media*. T. 11. 1, p. 3–19 (cf. p. 13).

Berlin, Team (2007). *Spirit of Berlin : An Autonomous Car for the DARPA Urban Challenge Hardware and Software Architecture* (cf. p. 14).

Bollella, Greg (2000). « The Real-Time Speci cation for Java ». In : *Computer* June, p. 47–54 (cf. p. 13).

Borkar, Shekhar (2005). « Designing reliable systems from unreliable components : The challenges of transistor variability and degradation ». In : *IEEE Micro* 25.6, p. 10–16 (cf. p. 56).

Borrelly, J-J et al. (1998). « The ORCCAD Architecture ». In : *The International Journal of Robotics Research* 17.4, p. 338–359 (cf. p. 6).

Brown, L (2006). « Linux Laptop Battery Life ». In : *Linux Symposium Volume One*, p. 127 (cf. p. 57).

Burns, Alan et B Littlewood (2010). « Reasoning About the Reliability of Multi-version, Diverse Real-Time Systems ». In : *2010 31st IEEE Real-Time Systems Symposium*, p. 73–81 (cf. p. 56).

Chanel P. Carvalho, Caroline, Florent Teichteil-Königsbuch et Charles Lesire (2013). « Multi-Target Detection and Recognition by UAVs Using Online POMDPs ». In : *Twenty-Seventh AAAI ...* 2013.July, p. 1381–1387 (cf. p. 14).

Chen, Min-Ih et Kwei-Jay Lin (1990). « Dynamic priority ceilings : A concurrency control protocol for real-time systems ». In : *Real-Time Systems* 2.4, p. 325–346 (cf. p. 27).

Chetto, Houssine, M Silly et T Bouchentouf (1990). « Dynamic scheduling of real-time tasks under precedence constraints ». In : *Real-Time Systems* 2.3, p. 181–194 (cf. p. 28).

Cucu-Grosjean, Liliana et al. (2012). « Measurement-based probabilistic timing analysis for multi-path programs ». In : *Proceedings - Euromicro Conference on Real-Time Systems*, p. 91–101 (cf. p. 56).

Dertouzos, M L (1974). *Control Robotics : the Procedural Control of Physical Processes," Information Processing 74* (cf. p. 25).

Dhouib, Saadia et al. (2012). « RobotML, a domain-specific language to design, simulate and deploy robotic applications ». In : *Lecture Notes in Computer Science (including subseries Lecture Notes in Artificial Intelligence and Lecture Notes in Bioinformatics)*. T. 7628 LNAI, p. 149–160 (cf. p. 18).

Dvorak, D et al. (2000). « Software architecture themes in JPL's Mission Data System ». In : *2000 IEEE Aerospace Conference. Proceedings (Cat. No.00TH8484)*. T. 7. October 1998, p. 259–268 (cf. p. 6).

Echeverria, Gilberto et al. (2012). « Simulating complex robotic scenarios with MORSE ». In : *Lecture Notes in Computer Science (including subseries Lecture Notes in Artificial Intelligence and Lecture Notes in Bioinformatics)* 7628 LNAI, p. 197–208 (cf. p. 108).

Feiler, Peter H, David P Gluch et John J Hudak (2006). « The Architecture Analysis & Design Language (AADL) : An Introduction ». In : CMU/SEI-2006-TN-011, pages (cf. p. 16).

Feiler, Peter H, David P Gluch, John J Hudak et Bruce A Lewis (2004). « Embedded System Architecture Analysis Using SAE AADL ». In : *Managing*, p. 1–45 (cf. p. 16).

Freitas, Gustavo et al. (2012). « A practical obstacle detection system for autonomous orchard vehicles ». In : *IEEE International Conference on Intelligent Robots and Systems*. IEEE, p. 3391–3398 (cf. p. 13).

Gat, Erann (1998). « On three-layer architectures ». In : *Artificial intelligence and mobile robots : case ...* (Cf. p. 6, 7, 111).

Georgas, John C et Richard N Taylor (2007). « An Architectural Style Perspective on Dynamic Robotic Architectures ». In : *Proceedings of the IEEE Second International Workshop on Software Development and Integration in Robotics (SDIR 2007), Rome, Italy* (cf. p. 6).

Gobillot, Nicolas, David Doose, Christophe Grand et al. (2015). « Real-time analysis of robotic software architectures ». In : *Control Architecture of Robots*. Paris, France (cf. p. 123).

Gobillot, Nicolas, David Doose, Charles Lesire et al. (2015). « Periodic state-machine aware real-time analysis ». In : *Emerging Technologies and Factory Automation*, p. 8 (cf. p. 122).

Gobillot, Nicolas, Charles Lesire et David Doose (2013). « A Component-Based Navigation-Guidance-Control Design Pattern for Mobile Robots ». In : *ICRA Workshop on Software Development and Integration for Robotics (SDIR)*, p. 1–3 (cf. p. 123).

— (2014). « A Modeling Framework for Software Architecture Specification and Validation ». In : *Simulation, Modeling, and Programming for Autonomous Robots*, p. 303–314 (cf. p. 41, 121).

Grisetti, Giorgio, Cyrill Stachniss et Wolfram Burgard (2007). « Improved techniques for grid mapping with Rao-Blackwellized particle filters ». In : *IEEE Transactions on Robotics* 23.1, p. 34–46 (cf. p. 112).

Hansen, Jeffery, Scott Hissam et Gabriel Moreno (2009). « Statistical-Based WCET Estimation and Validation ». In : *Wcet*. January, p. 123–133 (cf. p. 56).

Hart, P E, N J Nilsson et B Raphael (1968). « A Formal Basis for the Heuristic Determination of Minimum Cost Paths ». In : *IEEE Transactions on Systems Science and Cybernetics* 4.2, p. 100–107 (cf. p. 114).

Joseph, M et P Pandya (1986). « Finding response times in a real-time system ». In : *The Computer Journal* 29.5, p. 390–395 (cf. p. 86).

Jung, Min Yang et al. (2013). « Lessons Learned from the Development of Component-Based Medical Robot Systems ». In : *Journal of Software Engineering for Robotics* 5.In Review, p. 25–41 (cf. p. 13).

Kendoul, Farid (2012). « Survey of Advances in Guidance, Navigation, and Control of Unmanned Rotorcraft Systems ». In : *Journal of Field Robotics* 29.2, p. 315–378 (cf. p. 111).

Kim, Jong Hyuk, Salah Sukkarieh et Stuart Wishart (2006). « Real-time navigation, guidance, and control of a UAV using low-cost sensors ». In : *Springer Tracts in Advanced Robotics* 24, p. 299–309 (cf. p. 111).

Klotzbücher, Markus et Herman Bruyninckx (2012). « Coordinating Robotic Tasks and Systems with rFSM Statecharts ». In : *Journal of Software Engineering in Robotics* 1.January, p. 28–56 (cf. p. 44).

Konolige, Kurt et Karen Myers (1997). « The Saphira architecture : A design for autonomy ». In : *Journal of experimental & theoretical artificial intelligence* 9.2 - 3, p. 215–235 (cf. p. 6).

Lacroix, Simon et al. (2002). « Autonomous Rover Navigation on Unknown Terrains : Functions and Integration ». In : *The International Journal of Robotics Research* 21.10-11, p. 917–942 (cf. p. 111).

Leung, Joseph Y-T et Jennifer Whitehead (1982). « On the complexity of fixed-priority scheduling of periodic, real-time tasks ». In : *Performance Evaluation* 2.4, p. 237–250 (cf. p. 23).

Liu, L, C James et W Layland (1973). « Scheduling Algorithms for Multiprogramming in a Hard- Real-Time Environment Scheduling Algorithms for Multiprogramming ». In : *Journal of the Association for Computing Machinery* 20.1, p. 46–61 (cf. p. 23–25, 70).

Lu, Yue et al. (2012). « A Statistical Response-Time Analysis of Real-Time Embedded Systems ». In : *2012 IEEE 33rd Real-Time Systems Symposium*, p. 351–362 (cf. p. 125).

Mallet, Anthony, Cédric Pasteur et Matthieu Herrb (2010). « GenoM3 : Building middleware-independent robotic components ». In : *International Conference on Robotics and Automation (ICRA)*. Anchorage, AK, USA, p. 4627–4632 (cf. p. 12, 101).

Mallet, Anthony, Cédric Pasteur, Matthieu Herrb et al. (2010). « GenoM3 : Building middleware-independent robotic components ». In : *Proceedings - IEEE International Conference on Robotics and Automation*. Anchorage, AK, USA : Ieee, p. 4627–4632 (cf. p. 43).

Martin, S et P Minet (2006). « Schedulability analysis of flows scheduled with FIFO : application to the expedited forwarding class ». In : *Parallel and Distributed Processing Symposium, 2006. IPDPS 2006. 20th International*, 8 pp. (Cf. p. 23).

McGann, Conor et al. (2008). « A deliberative architecture for AUV control ». In : *Proceedings - IEEE International Conference on Robotics and Automation*. Pasadena, California, USA : IEEE, p. 1049–1054 (cf. p. 111).

McGann, C et al. (2008). « A deliberative architecture for AUV control ». In : *International Conference on Robotics and Automation (ICRA)*. Pasadena, California, USA, p. 1049–1054 (cf. p. 6).

Mok, Aloysius (1983). *Fundamental design problems of distributed systems for the hard-real-time environment* (cf. p. 24).

Mok, Aloysius et Deji Chen (1997). « A multiframe model for real-time tasks ». In : *IEEE Transactions on Software Engineering* 23.10, p. 635–645 (cf. p. 29).

Muscettola, Nicola et al. (2002). « IDEA : Planning at the Core of Autonomous Reactive Agents ». In : *Iwpss* 1.May, p. 7 (cf. p. 6).

Passama, Robin (2006). « Conception et développement de contrôleurs de robots-Une méthodologie bas{\'e}e sur les composants logiciels ». Thèse de doct. (cf. p. 39).

Quigley, Morgan et al. (2009). « ROS : an open-source Robot Operating System ». In : *Icra*. T. 3. Figure 1. Kobe, Japan, p. 5 (cf. p. 2, 12, 101).

Santinelli, Luca et al. (2014). « On the Sustainability of the Extreme Value Theory for WCET Estimation ». In : *the 14th International Workshop on Worst-Case Execution Time (WCET) Analysis*. Wcet, p. 21–30 (cf. p. 56).

Schlegel, Christian (2006). « Communication Patterns as Key Towards Component-Based Robotics ». In : *Journal of Advanced Robotic Systems* 3.1, p. 49–54 (cf. p. 42).

Schlegel, Christian et T Hassler (2009). « Robotic software systems : From code-driven to model-driven designs ». In : *International Conference on Advanced Robotics*, p. 1–8 (cf. p. 39).

Selic, Bran et Sébastien Gérard (2013). *Modeling and Analysis of Real-Time and Embedded Systems with UML and MARTE : Developing Cyber-Physical Systems*. Elsevier (cf. p. 16).

Sha, Lui, Ragunathan Rajkumar et John P Lehoczky (1990). « Priority inheritance protocols : An approach to real-time synchronization ». In : *IEEE Transactions on Computers* 39.9, p. 1175–1185 (cf. p. 27).

Shiraishi, Shin (2010). « An AADL-Based Approach to Variability Modeling of ». In : *Model Driven Engineering Languages and Systems* 6394 LNCS.PART 1, p. 346–360 (cf. p. 16).

Smits, Ruben et al. (2009). « ITASC : A tool for multi-sensor integration in robot manipulation ». In : *Lecture Notes in Electrical Engineering* 35 LNEE, p. 235–254 (cf. p. 14).

Soetens, Peter et Herman Bruyninckx (2005). « Realtime hybrid task-based control for robots and machine tools ». In : *International Conference on Robotics and Automation (ICRA)*. Barcelona, Spain (cf. p. 13).

Spuri, Marco (1996). *Analysis of Deadline Scheduled Real-Time Systems*. Rapp. tech., p. 34 (cf. p. 70).

Stigge, Martin et al. (2011). « The digraph real-time task model ». In : *Real-Time Technology and Applications - Proceedings*, p. 71–80 (cf. p. 29, 30).

Szyperski, Clemens, Dominik Gruntz et Murer Stephan (202). *Component Software : Beyond Object-Oriented Programming*. Reading, MA : Addison-Wesley, p. 625 (cf. p. 41).

Tchidjo Moyo, Noel et al. (2010). « On schedulability analysis of non-cyclic generalized multiframe tasks ». In : *Proceedings - Euromicro Conference on Real-Time Systems*, p. 271–278 (cf. p. 29).

Varró, Dániel (2002). « A Formal Semantics of UML Statecharts by Model Transition Systems ». In : *Graph Transformation* 2505.Graph Transformation, p. 378–392 (cf. p. 44).

Volpe, Richard et al. (2001). « The CLARAty architecture for robotic autonomy ». In : *2001 IEEE Aerospace Conference Proceedings*. T. 1. Big Sky, MT, USA, p. 1/121–1/132 (cf. p. 10).

Wilhelm, Reinhard et al. (2008). « The worst-case execution-time problem-overview of methods and survey of tools ». In : *ACM Transactions on Embedded Computing Systems* 7.3, p. 1–53 (cf. p. 55).

Publications

Conférences internationales

Gobillot, Nicolas, David Doose, Charles Lesire et al. (2015). « Periodic state-machine aware real-time analysis ». In : *Emerging Technologies and Factory Automation*, p. 8.

Gobillot, Nicolas, Charles Lesire et David Doose (2014). « A Modeling Framework for Software Architecture Specification and Validation ». In : *Simulation, Modeling, and Programming for Autonomous Robots*, p. 303–314.

Ateliers

Gobillot, Nicolas, Charles Lesire et David Doose (2013b). « A Component-Based Navigation-Guidance-Control Design Pattern for Mobile Robots ». In : *ICRA Workshop on Software Development and Integration for Robotics (SDIR)*, p. 1–3.

Journées francophones

Gobillot, Nicolas, David Doose, Christophe Grand et al. (2015). « Real-time analysis of robotic software architectures ». In : *Control Architecture of Robots*. Paris, France.

Gobillot, Nicolas, Charles Lesire et David Doose (2013a). « A Component-Based Navigation-Guidance-Control Architecture for Mobile Robots ». In : *Control Architectures of Robots*. Angers, France, p. 1–12.

Gobillot, Nicolas, Alessandra Melani et al. (2015). « Building System Awareness : Cache Characterization through Probabilities ». In : *Formalisation des Activités Concurrentes (FAC)*, p. 12.

Résumé — Un système robotique est un système complexe, à la fois d'un point de vue matériel et logiciel. Afin de simplifier la conception de ces machines, le développement est découpé en modules qui sont ensuite assemblés pour constituer le système complet. Cependant, la facilité de conception de ces systèmes est bien souvent contrebalancée par la complexité de leur mise en sécurité, à la fois d'un point de vue fonctionnel et temporel.

Il existe des ensembles d'outils et de méthodes permettant l'étude d'ordonnançabilité d'un système logiciel à base de tâches. Ces outils permettent de vérifier qu'un système de tâches respecte ses contraintes temporelles. Cependant ces méthodes d'analyse considèrent les tâches comme des entités monolithiques, sans prendre en compte la structure interne des tâches, ce qui peut les rendre trop pessimistes et non adaptées à des applications robotiques.

Cette étude consiste à prendre en compte la structure interne des tâches dans des méthodes d'analyse d'ordonnançabilité. Cette thèse montre que le découpage de tâches monolithiques permet d'améliorer la précision des analyses d'ordonnancement. De plus, les outils [1] issus de ces travaux ont été expérimentés sur un cas d'application de robotique mobile autonome.

Mots clés : Robotique, Validation, Temps réel, Architecture, Logiciel, Sûreté

Abstract — A robot is a complex system combining hardware and software parts. In order to simplify the robot design, the whole system is split in several separated modules. However, the complexity of the functional and temporal validation to improve the safety counterweights the robot design simplicity.

We can find scheduling analysis tools for task-based software. These tools are used to check and validate the schedulability of the tasks involved in a software, run on a specific hardware. However, these methods considers the tasks as monolithic entities, without taking into account their internal structure. The resulting analyses may be too much pessimistic and therefore not applicable to robotic applications.

In this work, we have modeled the internal structure of the tasks as state-machines and used these state-machines into the schedulability analysis in order to improve the analysis precision. Moreover, the tools [1] developed during this work have been tested on real robotic use-cases.

Keywords : Robotics, Validation, Real-time, Architecture, Software, Safety

2 avenue Edouard Belin
31055 Toulouse, France

1. https://forge.onera.fr/projects/mauve