Faciliter le développement des applications de robotique

Selma Kchir

Thèse de doctorat

de l'Université Pierre & Marie Curie

Spécialité

Informatique

Ecole Doctorale Informatique, Télécommunications et Électronique (Paris)

Présentée par

Selma Kchir

Pour obtenir le grade de

DOCTEUR de l'UNIVERSITÉ PIERRE ET MARIE CURIE

Sujet de la thèse

Faciliter le développement des applications de robotique

Date de soutenance: le *26/06/2014*

Devant le jury composé de:

M.Xavier Blanc	Professeur à l'Université Bordeaux 1	Examinateur
M.Noury Bouraqadi	Maître de conférences HDR à l'Ecole des mînes de Douai	Rapporteur
M.Jacques Malenfant	Professeur à l'Université Pierre et Marie Curie	Examinateur
M.Bernard Pottier	Professeur à l'Université de Bretagne Occidentale	Examinateur
M.Amar Ramdane-Chérif	Professeur à l'Université de Versailles Saint Quentin en Yvelines	Rapporteur
M.Serge Stinckwich	Maître de conférences à l'Université Caen	Co-Encadrant
M.Tewfik Ziadi	Maître de conférences à l'Université Pierre et Marie Curie	Co-Encadrant
M.Mikal Ziane	Maître de conférences HDR à l'Université Paris Descartes	Directeur

Table des matières

Table des figures

Liste des tableaux

Introduction générale

1 Contexte et problèmatique

Les roboticiens sont confrontés à plusieurs difficultés pour le développement de leurs applications en raison de l'hétérogénéïté des concepts de la robotique. En effet, dans le cas de la robotique mobile les roboticiens doivent maîtriser les détails liés aux moyens de locomotion du robot, à sa morphologie ainsi qu'à ses capteurs. De plus, la variabilité du matériel rend les applications de robotique fragiles : un changement du matériel dans une application déjà développée impliquerait la réécriture du code de celle-ci.
Il faudrait alors découpler le code de l'application à travers des abstractions de haut niveau afin de la rendre plus stable vis-à-vis des changements du matériel.
Pour répondre à ce constat de variabilité du matériel, certains middleware de robotique tels que ROS [10], MIRO [11], PyRO [12] et Player [13] ont proposé des abstractions du matériel pour permettre une montée en niveau d'abstractions par rapport aux détails du matériel. Ces dernières permettent d'encapsuler des données spécifiques au matériel afin de fournir des données de plus haut niveau. Cependant, ces abstractions restent de bas niveau et ne permettent pas une isolation de certains changements dans le matériel. Il serait alors plus simple pour les roboticiens de manipuler, à travers des outils, des concepts qu'ils ont l'habitude d'utiliser et qui soient de plus haut niveau que ceux proposés par les middleware.

Dans cette optique qui vise à définir des abstractions de haut niveau pour les applications de robotique et à faciliter leur développement, les langages de modélisation spécifiques au domaine (en anglais DSML : Domain Specific Modeling Language) offrent à travers des notations appropriées et des abstractions, une puissance expressive axée sur, et généralement limitée à, un domaine particulier [14].

Ces abstractions sont définies par les experts du domaine d'application réduisant ainsi le fossé sémantique entre les développeurs et les experts du domaine afin de faciliter le développement des applications du domaine.
L'utilisation de l'ingénierie dirigée par les modèles (IDM) (en anglais Model Driven Engineering (MDE)) [15] dans ce contexte permettrait de gérer les problèmes de dépendance de bas niveau (c.-à-d. variabilité du matériel et des plateformes) à travers des modèles stables basés sur des abstractions du domaine. De plus, la génération automatique de code à partir de ces modèles vers plusieurs plateformes cibles offrirait un gain de temps et une facilité pour le développement des applications de robotique.
Plusieurs DSML pour la robotique ont été définis tels que BCM [3], SmartSoft [16] et ORCCAD [4] mais ils s'intéressent essentiellement aux aspects qui relèvent de l'architecture d'une application (architecture à base de composants) et à la communication entre ses composants. Les abstractions utilisées dans ces DSML sont insuffisantes car elles ne permettent pas de représenter tous les concepts d'une application tels qu'ils sont manipulés par les roboticiens. Par ailleurs, ils ne définissent pas d'abstractions pour le matériel.
Il faudrait alors convenir d'un ensemble d'abstractions stables qui permettraient de représenter les concepts d'une application de robotique afin de faciliter son développement et garantir son indépendance par rapport aux plateformes cibles. Ces concepts peuvent être des concepts de haut niveau comme ceux proposés par les DSML. Cependant ces derniers manquent souvent de sémantique opérationnelle car ils représentent souvent des concepts généraux du domaine. Il faudrait alors définir des abstractions de haut niveau qui permettent , indépendamment du matériel utilisé, d'obtenir des informations sur l'environnement du robot et de définir des actions de haut niveau d'une part et d'encapsuler les détails algorithmiques d'une application d'une autre part.

2 Contributions

Cette thèse tente d'apporter des solutions aux problèmes présentés précédemment. Dans cette optique, nous avons contribué au développement d'un DSML pour la robotique (RobotML) dans le cadre d'un projet ANR, appelé PROTEUS. Les abstractions fournies par RobotML se basent sur une ontologie du domaine de la

robotique basée sur les connaissances des experts du domaine. Des outils ont été développés autour de RobotML permettant la modélisation de scénarios de robotique et une génération automatique de code vers une ou plusieurs plateformes cibles.

- Il devient alors facile pour un expert en robotique et même pour des simples utilisateurs débutants en robotique de représenter les scénarios de leurs applications.
- Les modèles représentés avec RobotML sont stables et indépendants des plateformes cibles.
- La génération de code à partir de RobotML se fait vers une ou plusieurs plateformes hétérogènes (c.-à-d. simulateurs ou robots réels d'une part et middleware d'une autre part) et permet d'obtenir une application compilable ainsi que les artefacts nécessaires pour son exécution.

Cependant, les expériementations basées sur RobotML ont montré que les abstractions du domaine définies dans l'ontologie sont insuffisantes. En effet, les abstractions de l'ontologie sont basées sur les connaissances du domaine et non pas sur plusieurs itérations appliquées sur des exemples concrets. Par conséquent, elles manquent de sémantique opérationnelle et sont trop générales pour pouvoir être utilisées pour la définition d'abstractions du matériel par exemple. Il faudrait alors partir d'exemples concrets et appliquer une approche complémentaire pour enrichir les abstractions du domaine avec des abstractions a posteriori.
Nous avons donc proposé une approche qui, à partir de la description d'une tâche de robotique et des hypothèses sur l'environnement et sur les capacités du robot, produit :

- un ensemble d'abstractions non algorithmiques représentant des requêtes sur l'environnement y compris le robot ou des actions de haut niveau ;
- un ensemble d'abstractions algorithmiques représentant un ensemble d'instructions permettant de réaliser une sous-tâche de la tâche à développer ;
- un algorithme générique configurable défini en fonction de ces abstractions.

Ainsi, l'impact du changement du matériel et des stratégies définies dans les sous-tâches n'est pas très important. Il suffit d'adapter l'implantation de ces abstractions sans avoir à modifier l'algorithme générique.

3 Plan de la thèse

Cette thèse s'organise comme suit : Le chapitre 1 présente tout d'abord le contexte de ce travail : la variabilité dans le domaine de la robotique. Nous présenterons les différentes catégories des capteurs, des effecteurs, des architectures de robotique et des algorithmes de robotique. Nous dresserons ensuite dans le chapitre 2 un état de l'art sur les abstractions proposées par certains des middlewares de robotique existants pour répondre au problème de la variabilité du matériel. Nous présenterons ensuite dans le chapitre 3 les DSML de robotique proposés pour faciliter le développement des applications de robotique et garantir leur indépendance par rapport aux plateformes cibles. Nous présenterons notre DSML pour la robotique (RobotML) ainsi que la génération de code vers un middleware de robotique (OROCOS) et un scénario que nous avons utilisé pour la validation de nos travaux dans le chapitre 4. Notre approche pour la définition des abstractions pertinentes et d'un algorithme générique à partir de la description d'une tâche de robotique ainsi que son application sur une famille d'algorithmes de navigation sera présentée dans le chapitre 5. Nous concluerons cette thèse par un rappel de nos contributions et des discussions sur les perspectives.

Première partie

Etat de l'art

La variabilité dans le domaine de la robotique

1 Introduction

La robotique mobile désigne l'étude des robots à base mobile qui, par opposition aux robots marcheurs, aux robots manipulateurs ou encore aux robots aériens, sont des robots ayant pour moyen de locomotion des roues. Les définitions du terme *robot* dans la littérature sont diverses, quelquefois très générales comme la définition de Brady [17] "*[...] une connexion intelligente entre la perception et l'action*". Arkin [1], quant à lui, a donné une définition plus spécifique "*Un robot intelligent mobile est une machine capable d'extraire des informations sur son environnement et d'utiliser les connaissances sur son monde afin de se déplacer de manière significative et en toute sécurité*". Par sa définition, Arkin explique qu'il existe une connexion entre la perception et l'action ; deux concepts fondamentaux en robotique qui varient d'un robot à l'autre. Cette connexion n'est autre que le système de contrôle du robot qui définit un ensemble de fonctions permettant au robot d'atteindre le but de sa mission. La perception, l'action et le système de contrôle forment une application de robotique. On peut parler alors de plusieurs types de variabilités : la variabilité liée à la perception, à l'action et au contrôle du robot qui concerne les aspects architecturaux d'une application. Dans ce chapitre, nous nous proposons de présenter les concepts de la robotique que nous allons utiliser tout au long de cette thèse ainsi que les principaux types de variabilité en robotique.

2 Définitions

En raison de la diversité des définitions présentes dans la littérature, nous nous proposons dans cette section de présenter les définitions des concepts que nous utiliserons tout au long de cette thèse. Nous nous plaçons dans le contexte de la **robotique mobile non humanoïde** où les robots sont équipés de capteurs et d'effecteurs afin d'intéragir avec leur environnement.

Definition 1 (Robot mobile autonome)**.** *Un robot mobile autonome est un robot intelligent capable d'exécuter une tâche sans l'intervention des humains, il est capable de percevoir l'état de son environnement et de se déplacer dans celui-ci afin d'accomplir une tâche particulière (définition adaptée de la définition d'Arkin [1]).*

Definition 2 (Perception)**.** *La perception est l'extraction d'informations sensorielles, à partir des capteurs, indiquant l'état du robot dans son environnement.*

Definition 3 (Action)**.** *L'action est l'interprétation des données sensorielles ou de directives envoyées par le système de contrôle du robot en commandes envoyées aux effecteurs du robot.*

Definition 4 (Tâche)**.** *Une tâche est constituée d'un ensemble d'actions permettant au robot d'accomplir un but ou un sous-but.*

Definition 5 (Planification)**.** *La planification est l'analyse d'une séquence d'actions primitives afin de réduire l'écart entre l'état actuel du robot et le but qu'il doit atteindre. (définition adaptée de Nilsson [18] et du paradigme General Problem Solver [19]).*

Definition 6 (Système de contrôle)**.** *Le système de contrôle du robot est le système responsable de coordonner les parties perception et action du robot afin de réaliser une tâche particulière.*

Definition 7 (Paradigme)**.** *Un paradigme est une philosophie ou un ensemble d'hypothèses et/ou de techniques qui caractérisent une approche pour une classe de problèmes [20].*

Definition 8 (Architecture)**.** *Une architecture de robotique est une discipline consracrée à la conception de robots hautement spécifiques et individuels à partir d'une collection de modules logiciels communs [1]. Une architecture est un pattern pour l'implémentation d'un paradigme [20].*

Definition 9 (Middleware)**.** *Un middleware est une couche d'abstraction entre le système d'exploitation et l'application [21].*

Definition 10 (Variabilité)**.** *La variabilité désigne l'étendue des différentes catégories que peut prendre un concept. En robotique, on trouve de la variabilité dans les capteurs, dans les effecteurs, dans les algorithmes, dans les architectures, etc.*

3 La variabilité dans le domaine de la robotique

Dans cette section, nous présentons la variabilité dans les concepts présentés dans la section précédente. La variabilité désigne la diversité des catégories ou des classifications pour un concept donné (une tâche, un capteur, un effecteur, un environnement, etc).

3.1 La variabilité des capteurs

La perception, comme nous l'avons définie précédemment, est l'extraction d'informations de l'environnement du robot à travers ses capteurs. Le choix des capteurs dépend de la nature des informations dont le robot a besoin pour accomplir sa tâche. Les capteurs sont des dispositifs qui répondent à un signal ou à un stimulus. Un stimulus est une quantité, une propriété ou une condition détectée et convertie en un signal électrique [22]. À la différence des transducteurs qui convertissent un type d'énergie en un autre type d'énergie, les capteurs convertissent une énergie en un signal électrique. Dans la littérature, les capteurs sont généralement regroupés selon deux principales catégories : proprioceptifs ou exteroceptifs.

- Les capteurs **proprioceptifs** mesurent l'état interne du robot. Comme par exemple, la vitesse du moteur, le niveau de batterie, etc.
- Les capteurs **exteroceptifs** récupèrent les informations à partir de l'environnement du robot. Par exemple, l'intensité de la lumière, une mesure de distance, etc.

Les capteurs sont également classés relativement à la nature de leur fonctionnement :

- Les capteurs **passifs** mesurent l'énergie présente dans l'environnement, par exemple les microphones, les sondes de température, etc.
- Les capteurs **actifs** émettent une énergie dans l'environnement et mesurent la réaction de l'environnement à celle-ci.

Une autre classification des capteurs dans la littérature dépend des données que les capteurs peuvent mesurer (stimulus), leurs spécifications, le mécanisme de conversion qu'ils utilisent et le matériel à partir duquel ils sont conçus [22]. Par exemple dans le cas des capteurs à portée (appelés aussi capteurs à rayons), nous notons une variabilité par rapport à la portée des capteurs, aux nombres de rayons, à la largeur du (ou des) rayons dont ils disposent, au champs de vision (*Field of view*) des capteurs, etc.

Classification	Capteurs	Type
Capteurs de localisation	GPS	exteroceptif actif
	Odométrie	proprioceptif actif
Capteurs à rayons	Laser	exteroceptif actif
	Infrarouge	exteroceptif actif
	Ultrasonic	exteroceptif actif
Capteurs basés sur la vision	Caméra	exteroceptif passif
Capteurs Moteur/Roues	encodeurs inductifs	proprioceptif passif
Capteurs tactiles	Bumpers, Contact switches	exteroceptif passif

Table I.1 – Classification des capteurs inspirée de [8] et de [9]

Le tableau I.1 montre quelques exemples de capteurs proprioceptifs et exteroceptifs. Typiquement, le cas du GPS et de l'odométrie qui tous deux, servent à la localisation du robot mais ne sont pas de la même nature. La localisation consiste à indiquer la position du robot et son orientation.

– l'Odométrie : Les roues motrices d'un robot sont habituellement associées à un servomoteur. Celui-ci est équipé d'un dispositif de mesure de rotation, il s'agit d'un capteur proprioceptif appelé odomètre. L'odométrie est le moyen de localisation le plus simple qui permet d'estimer la position d'un robot mobile en mouvement. À une position donnée correspondent une multitude d'orientations possibles du robot. L'orientation du robot ne peut s'obtenir qu'en connaissant son orientation de départ ainsi que son évolution sur sa trajectoire. L'utilisation de l'odométrie est basée sur l'hypothèse de roue sans glissement (contrainte non holonomique que nous présenterons dans la section suivante) et sur la supposition que les paramètres géométriques du robot sont parfaitement connus.

– Global Positionning System (GPS) : est un système de géolocalisation qui effectue des émissions synchronisées dans le temps à des satellites. A l'aide des messages émis aux satellites, le récepteur peut calculer sa position. Le principe de calcul de la position s'appuye sur le principe de la triangulation (calcul de la distance par rapport à 3 satellites). Néanmoins, des imprécisions de la mesure du temps peuvent se produire résultant du décalage entre les horloges des satellites (très précises) et les horloges des récepteurs (moins précises). Un quatrième satellite est donc nécessaire afin d'assurer la robustesse de la mesure. Le GPS permet une localisation approximative (à quelques mètres près). En robotique, une méthode différentielle est utilisée pour obtenir des résultats plus précis. La localisation se fait alors à l'aide de deux récepteurs dont l'un est statique et positionné avec précision dans l'environnement. La précision devient alors meilleure. L'utilisation du GPS impose la navigation à l'extérieur en raison de la présence de satellites.

Les applications de robotique sont implémentées en fonction des capteurs qu'elles utilisent. La variabilité des capteurs rend ces applications fragiles aux changements du matériel. D'un point de vue conceptuel, séparer l'acquisition des données du traitement de celles-ci est une étape importante pour assurer la pérennité et la consistence des applications de robotique.

3.2 La variabilité des effecteurs

Un robot agit sur son environnement à travers ses effecteurs. Dans la littérature, un effecteur (appelé aussi actionneur), à l'opposé d'un capteur, convertit un signal électrique en une énergie non électrique ou un phénomène physique [22]. Par exemple, un moteur convertit un signal en une action mécanique. L'étude détaillée des effecteurs, qui relève à la fois de la cinématique, de la mécanique et l'électronique ne sera pas présentée ici. Nous nous intéresserons dans cette section aux mouvements des robots mobiles.
Les robots mobiles sont généralement équipés par des roues. Chaque roue contribue au mouvement du robot et impose des contraintes sur le mouvement de celui-ci. Par conséquent, les contraintes définies sur les roues combinent les contraintes définies sur chaque roue individuelle [8]. Les mouvements des robots sont traditionnellement classés en deux catégories : les robots **holonomes** et les robots **non-holonomes** [23].

Les robots de type holonomes n'ont pas de restriction sur le déplacement contrairement aux robots de type non-holonomes qui possèdent des contraintes cinématiques sur leurs mouvements. Lorsque le contact entre le robot et le sol est ponctuel et que les roues sont indéfomables, celà se traduit mathématiquement par une vitesse nulle entre le sol et la roue. Il s'agit de contraintes de roue sans glissement. Ces contraintes sont des contraintes non holonomes. L'existence de contraintes non-holonomes implique que le robot ne peut pas effectuer certains mouvements instantanément, il doit manoeuvrer. Les robots mobiles à roues sont typiquement des systèmes non-holonomes.
Cette contrainte représente aussi une variabilité car la vitesse de rotation ou d'avancement des effecteurs diffère non seulement selon l'environnement dans lequel se trouve le robot mais aussi selon le chemin que ce dernier doit suivre.
De même que pour les capteurs, les applications de robotique sont généralement écrites en fonction des effecteurs du robot. Par conséquent, un découplage du code de l'application des détails des effecteurs du robot rendrait les applications résistantes aux changements des robots plus généralement et des effecteurs plus particulièrement.

3.3 La variabilité dans les architectures de robotique

Le but de choisir une architecture de robotique est de rendre la programmation d'un robot plus facile, plus flexible et plus sûre [24]. Les architectures de robotique permettent de définir la façon dont le robot doit agir de façon à satisfaire le but de sa mission. Dans cette section, nous présentons les différentes architectures définies dans la littérature.

3.3.1 Les architectures délibératives

Les architectures de robotique ont apparu à la fin des années 60 avec le **paradigme hiérarchique (*sense-plan-act*)** [18] (perception - planification - action) : le robot pense d'abord puis agit. La perception dans ce cas consiste à la traduction des informations sensorielles et cognitives en un modèle du monde [18, 25, 26]. À partir de ce modèle, le planificateur détermine quelles sont les actions à effectuer et renvoie les commandes à exécuter aux effecteurs. Le schéma de cette architecture est donné par la figure I.1. Parmi les architectures hiérarchiques/délibératives,

nous citons NASREM (The NASA Standard Reference Model for Telerobot Control System) [27] développée par la NASA au milieu des années 80.

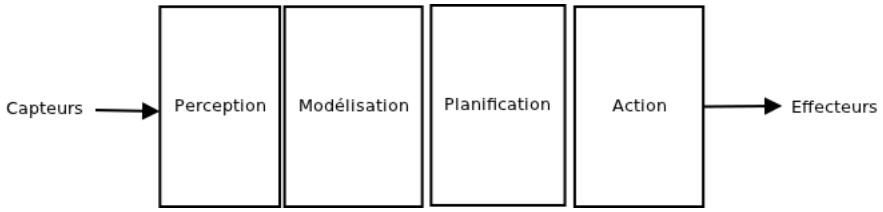

Figure I.1 – Architecture verticale - paradigme hiérarchique (adaptée de [1])

L'architecture délibérative a été adoptée dans les applications de robotique jusqu'au milieu des années 80 lorsqu'il a été constaté que le processus de planification est un processus complexe et non adapté à un environnement dynamique. En effet, le robot reste bloqué jusqu'à la réception des directives du module de planification. De plus, l'exécution d'un plan sans la prise en compte des changements qui peuvent surgir dans l'environnement rend cette architecture non adaptée à un environnement dynamique. C'est pour cette raison que les architectures réactives ont ensuite été définies afin de répondre aux situations où le robot doit agir rapidement.

3.3.2 Les architectures réactives

À l'opposé du paradigme Sense-Plan-Act, les architectures réactives s'appuient sur le **paradigme réactif *Sense-Act*** (Perception - Action) : le robot ne pense pas mais agit. Un système réactif *associe étroitement la perception à l'action sans l'utilisation de représentations abstraites ou d'historique du temps* [1]. Le robot ne se base plus sur le modèle de son monde mais uniquement sur des collections de conditions/actions définies à partir des données des capteurs. Cette architecture est adaptée aux environnements où le robot doit agir rapidement, typiquement dans le cas des environnements dynamiques.

Les **architectures basées sur le comportement** ont également été définies pour pallier aux problèmes des paradigmes hiérarchiques. Ces architectures sont souvent confondues avec les architectures réactives dans la littérature, notamment dans [28]. Cette confusion découle de l'utilisation des architectures basées sur le comportement de comportements réactifs dans la plupart des cas (même si la définition d'un comportement est plus sophistiquée que la définition d'une action dans les architectures réactives [29]). Un comportement est défini comme une activité primitive [30]

qui prend en entrée les données des capteurs et retourne une action à effectuer (principe de stimulus/réponse). Il peut être exprimé avec un triplet (S, R, β) où S est le stimulus, R est la réponse et β est le lien $S- > R$ [1]. La définition et l'extraction de ces comportements ont fait l'objet de plusieurs travaux et plusieurs approches ont été définies notamment l'approche éthologique (basée sur l'observation des comportements des animaux) [30, 31], l'approche guidée par les expérimentations (consistant à définir un comportement basique et à ajouter d'autres comportements ou à raffiner les comportements définis) [32] et l'approche dirigée par les buts, etc.

Afin d'assember les comportements, plusieurs approches ont été définies dont l'approche compétitive. Dans l'approche compétitive, on distingue 3 approches différentes :

- La coordination basée sur la priorité [33] : Le système d'arbitrage sélectionne la réponse envoyée par le comportement ayant la priorité la plus élevée.
- La coordination basée sur le vote [34] : Chaque comportement retourne un vote à une réponse et la réponse ayant le plus de votes est exécutée.
- La coordination basée sur la sélection des actions [35] : Tous les comportements retournent leurs réponses à l'arbitre qui se charge de sélectionner la réponse majoritaire.

Parmi les architectures basées sur le comportement, **l'architecture de subsomption(*subsumption architecture*)**, proposée par Brooks [30, 32, 33] vers la fin des années 80 est une architecture purement réactive. C'est une architecture incrémentale bottom-up (du bas vers le haut) allant du comportement le plus basique (placé en bas) vers le comportement ayant le degré de compétence le plus élevé (placé en haut). Chaque comportement est modulaire et est représenté par une machine à états finis. L'architecture de subsomption s'appuye sur la notion de parallélisme, elle permet l'exécution simultanée de plusieurs comportements qui interrogent les mêmes capteurs et agissent sur les mêmes effecteurs. Le mécanisme de coordination utilisé est un mécanisme compétitif ayant un système d'arbitrage basé sur la priorité à travers l'inhibition et la suppression comme le montre la figure I.2. Un comportement peut supprimer les sorties d'un autre comportement (cercle avec un S), dans ce cas le comportement dominant remplace les sorties de l'autre comportement par ses propores commandes. Un comportement peut également inhiber (cercle avec un I) les sorties d'un autre comportement en les bloquant.

L'architecture de subsomption ne permet pas l'utilisation de comportements indépendants. En effet, il y a une dépendance entre les comportements qui permet de

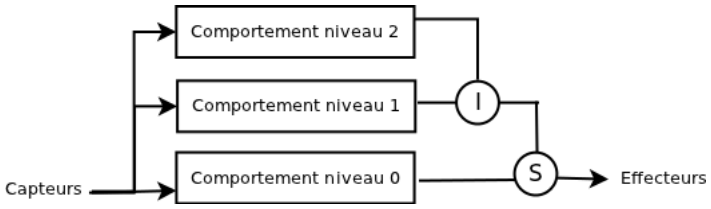

FIGURE I.2 – Architecture de subsomption

savoir comment les placer dans l'architecture (toujours en suivant un ordre de priorité). Malgré leurs avantages, les architectures basées sur le comportement ne sont pas adaptées à tous les types d'environnements. Il est difficile d'avoir la localisation du robot dans le modèle de son monde et ses connaissances par rapport à son monde sont limitées voire inexistantes [1].

3.3.3 Les architectures hybrides

L'architecture hybride a ensuite été définie afin de combiner les meilleurs aspects de l'architecture délibérative et de celle basée sur le comportement en utilisant **le paradigme Plan, Sense-Act** (Plannification, Perception-Action). La plannification est effectuée en une étape, la perception et l'action sont effectuées ensemble. L'architecture hybride comporte un système réactif pour le contrôle de bas niveau et un système délibératif de haut niveau pour la prise de décisions. L'architecture hybride a émergé au début des années 90 avec l'architecture AuRA [36] [37] (Autonomous Robot Architecture).

Chaque application de robotique utilise une architecture particulière. Dans certains systèmes existants, il est difficile de comprendre l'architecture utilisée. En effet, l'architecture et l'implémentation spécifique au domaine (à la plateforme, au langage, etc.) sont souvent liées brouillant ainsi le style architectural utilisé [24]. Il est donc important de séparer l'architecture des détails de l'implémentation afin de garantir la modularité des composants architecturaux.

3.4 La variabilité dans les algorithmes de robotique

Un autre type de variabilité concerne les algorithmes. Elle est souvent relative aux différentes variantes d'une famille d'algorithmes. Prenons le cas d'une tâche de navigation. La navigation est l'une des capacités les plus importantes des robots

mobiles. Le problème de la navigation consiste principalement à trouver un chemin sans collision pour se déplacer d'un point donné vers un autre point ou tout simplement à explorer un environnement tout en évitant les obstacles présents dans celui-ci. Il existe une grande variété de travaux permettant d'aborder le problème de la navigation. D'une façon générale, on peut distinguer deux types de navigation :

- La navigation dans un environnement connu (avec un modèle du monde). Le robot dispose du modèle de son monde et planifie par conséquent les actions à exécuter pour se déplacer dans celui-ci. On retrouve alors le paradigme hiérarchique ou le paradigme hybride.
- La navigation dans un environnement inconnu. Elle consiste à appliquer un ensemble de comportements réactifs dont on estime que l'enchaînement peut permettre au robot d'atteindre son but. Par exemple, le contournement d'obstacle, la poursuite de chemin, etc.

La navigation requiert des fonctionnalités que le robot doit être capable d'effectuer notamment, la localisation (qui permet au robot de se positionner dans son environnement s'il dispose du modèle de celui-ci ou par rapport à son but ou à sa position de départ) et l'évitement d'obstacles.
Il existe diverses stratégies pour l'évitement d'obstacles. Ces obstacles peuvent être de formes inconnues et placés de façon arbitraire dans l'environnement du robot.
Ainsi, même avec un ensemble d'hypothèses très simplifiées comme le cas de la famille Bug [38] (voir section 3.1 - chapitre 5), qui comporte environ vingt variantes, il n'est pas évident de comprendre ces algorithmes ou même de les comparer entre eux afin de décider quel algorithme est le plus efficace dans un environnement donné.
Si nous pouvions utiliser une seule implémentation d'un algorithme, la comparaison entre les variantes de cet algorithme devient simple et significative.

4 Conclusion

Nous avons présenté dans ce chapitre les principaux concepts de la robotique. Nous avons vu également les différents types de variabilité en robotique. Cette variabilité a un impact sur les applications de robotique les rendant ainsi

- **difficiles à développer** car le roboticien doit maîtriser les détails des capteurs, des effecteurs et de la tâche qu'il doit développer.

– **fragiles** car elles sont dépendantes des détails de bas niveau du matériel donc le changement de matériel a un impact important sur le code de l'application.
– **difficiles à comprendre** car elles sont confondues avec les détails d'implémentation des plateformes sous lesquelles elles sont développées.

Nous verrons dans le chapitre suivant comment certains middleware ont tenté de pallier la dépendance des applications de robotique au matériel en proposant des abstractions du matériel.

Les middleware de robotique

1 Introduction

Nous avons présenté dans le chapitre précédent la variabilité dans les concepts qui composent une application de robotique. Cette variabilité, notamment la variabilité du matériel, a orienté la conception de certains middleware de robotique pour (1) simplifier le processus de développement de ces applications (2) rendre ces applications indépendantes des détails du matériel (3) assurer la connexion entre les différents modules d'une application distribuée.
Bakken [39] définit un middleware comme étant : "*[...] une classe de technologies logicielles conçues pour aider à la gestion de la complexité et l'hétérogénéïté dans les systèmes distribués [...] Il fournit des abstractions de plus haut niveau que les APIs (Application Programming Interfaces) comme les sockets (fournies par le système d'exploitation)*".
Des études détaillées qui présentent les middleware de robotique, leurs architectures ainsi que leurs propriétés peuvent être trouvées dans [21, 40, 41].
Dans ce chapitre, nous ne présenterons pas l'aspect communication des middleware mais nous nous intéresserons particulièrement à l'abstraction du matériel dans les middlewares de robotique pour la gestion de la variabilité de bas niveau.
Nous présentons la description de certains des middlewares les plus utilisés en robotique : Player [13, 42], ROS [10, 43], MIRO [11] et PyRo [12, 44] ainsi que les abstractions du matériel qu'ils proposent et une discussion sur l'impact du changement du matériel illustrée sur un exemple d'évitement d'obstacles.

2 La gestion de la variabilité dans les middleware de robotique : abstraction du matériel

Selon [21, 41], idéalement les middleware de robotique devraient permettre la communication et l'interopérabilité des modules robotiques. Ces derniers étant implantés par des développeurs différents, il est difficile de les faire interagir entre eux. Un middleware doit donc supporter l'intégration des composants et des dispositifs de bas niveau du robot, assurer une utilisation efficace des ressources disponibles et supporter l'intégration avec les autres systèmes.
Comme nous l'avons mentionné précédemment, les aspects qui relèvent de la communication et des systèmes distribués ne font pas l'objet de ce travail. Nous nous intéressons essentiellement à la gestion de la variabilité du matériel. Certains middleware de robotique ont défini des données abstraites afin d'encapsuler les détails spécifiques au matériel utilisé. Dans cette section, nous présentons pour chaque middleware sa description, les abstractions du matériel proposées dans celui-ci ainsi qu'une discussion qui montre l'impact du changement du matériel sur un exemple d'évitement d'obstacles.

2.1 Player

2.1.1 Description

Le but de Player [13, 42] consiste à permettre l'exécution des mêmes applications aussi bien en simulation que sur des robots réels. Il fournit un framework de développement supportant différents périphériques matériels [1], des services communs requis pas les applications robotiques et transfère un contrôleur de la simulation à des robots réels avec le minimum d'effort possible [21]. Un programme client communique avec Player, qui s'exécute sur le robot, à l'aide d'une connexion de socket TCP pour le transfert de données. Player contient trois modèles :

– Le modèle dispositif (*Character device*). Les *character devices* sont des dispositifs qui fournissent et consomment des flots de données au fil du temps. C'est typiquement le cas des capteurs et des effecteurs. L'interface *Character device* définit des opérations (open, close, read, write, etc.) pour l'accès et le contrôle des capteurs et des effecteurs.

1. http://playerstage.sourceforge.net/doc/Player-2.1.0/player/supported_hardware.html

– Le modèle Interface/Pilote (*Interface/Driver*) : Le modèle interface/driver regroupe les dispositifs selon des fonctionnalités logiques de façon à ce que les dispositifs capables d'effectuer la même fonctionnalité apparaissent d'une façon similaire à l'utilisateur.
– Le modèle Client/serveur : Player supporte plusieurs connexions clientes aux dispositifs, créant ainsi des nouvelles possibilités pour la perception et le contrôle collaboratif.

2.1.2 Les abstractions du matériel dans Player

Les abstractions du matériel résident dans le modèle Interface/Driver. L'interface est une spécification du contenu des flots de données. Elle fournit un ensemble de messages communs à une certaine classe de dispositifs indépendamment de la plateforme utilisée. Le code qui implémente l'interface et qui convertit le format natif des commandes de l'API de communication en commandes requises par l'interface et inversement est appelé driver.
Le modèle *Character device* relie les flots en entrée du programme aux données des capteurs et les flots de sortie aux commandes des effecteurs. Les drivers sont spécifiques aux dispositifs ou à une famille de dispositifs d'un même "vendeur".
Player sépare les fonctionnalités logiques des détails d'implémentation des dispositifs. On peut dire alors que *Interface* correspond à la notion de classe abstraite regroupant l'ensemble des fonctionnalités d'un type spécifique de capteurs ou d'actionneurs et que *Driver* est une classe concrète.

```
/*Request and change the device's configuration. */
typedef struct player_ranger_config{
  double min_angle; /** Start angle of scans [rad]. May be unfilled. */
  double max_angle; /** End angle of scans [rad]. May be unfilled. */
  double angular_res; /** Scan resolution [rad]. May be unfilled. */
  double min_range; /** Minimum range [m].  */
  double max_range; /** Maximum range [m]. May be unfilled. */
  double range_res; /** Range resolution [m]. May be unfilled. */
  double frequency; /** Scanning frequency [Hz]. May be unfilled. */
} player_ranger_config_t;
/*The ranger device position, orientation and size. */
typedef struct player_ranger_geom{
  player_pose3d_t pose;  /** Device centre pose in robot CS [m, m, m, rad, rad, rad]. */
  player_bbox3d_t size; /** Size of the device [m, m, m]. */
  uint32_t element_poses_count; /** Number of individual elements that make up the device. */
  player_pose3d_t *element_poses; /** Pose of each individual element that makes up the device (in device CS). */
  uint32_t element_sizes_count; /** Number of individual elements that make up the device. */
  player_bbox3d_t *element_sizes; /** Size of each individual element that makes up the device. */
} player_ranger_geom_t;
```

Listing II.1 – Extrait de l'Interface RangeSensor de Player

L'interface `Ranger` (voir le listing II.1) fournit une API commune aux capteurs de distance. Les attributs *min_angle*, *max_angle*, *angular_res*, *min_range*, *max_angle*, *range_res* et *frequency* de la structure `player_ranger_config` permettent de configurer un capteur de distance donné. La structure `player_ranger_geom` permet de spécifier la géométrie du capteur à configurer. Dans ce qui suit, nous discuterons l'impact du changement du capteur de distance dans un algorithme d'évitement d'obstacles écrit avec Player.

2.1.3 Discussion

Le listing II.2 est un exemple d'algorithme d'évitement d'obstacles founi avec Player et programmé en C++ avec un capteur Sonar. Dans cet exemple, le robot avance dans son environnement. Si sa distance par rapport à l'obstacle le plus proche est inférieure à really_min_front_dist (lignes 9-13) il fait demi-tour sinon si cette distance est inférieure à min_front_dist (lignes 14-17), il s'arrête (ligne 27).
Supposons maintenant que l'on veuille remplacer dans cet exemple le capteur Sonar par un capteur Laser. Il faudrait remplacer *SonarProxy* par *LaserProxy* et décomposer ensuite les rayons contenus dans l'angle de perception du capteur Laser (si leur nombre le permet) en 5 sous-ensembles afin de regrouper les rayons du capteur Laser et les traiter comme les rayons du Sonar. Cela éviterait de modifier le code.

```
1     PlayerClient robot (gHostname, gPort);
2     Position2dProxy pp (&robot, gIndex);
3     SonarProxy sp (&robot, gIndex);
4     pp.SetMotorEnable (true);
5     double newspeed = 0.0f, newturnrate = 0.0f;
6     for(;;) {
7       robot.Read();
8       avoid = 0;    newspeed = 0.200;
9       if (avoid == 0) {
10          if((sp[2] < really_min_front_dist) || (sp[3] < really_min_front_dist) ||
11             (sp[4] < really_min_front_dist) || (sp[5] < really_min_front_dist)){
12              avoid = 50;    newspeed = -0.100;
13          }
14          else if((sp[2] < min_front_dist) || (sp[3] < min_front_dist) ||
15                  (sp[4] < min_front_dist) || (sp[5] < min_front_dist)){
16              newspeed = 0;  avoid = 50;
17          }
18      }
19      if(avoid > 0){
20        if((sp[0] + sp[1]) < (sp[6] + sp[7]))
21          newturnrate = dtor(-30);
22        else
23          newturnrate = dtor(30);
24        avoid--;
25      }
26      else
27        newturnrate = 0;
28      pp.SetSpeed(2 * newspeed, newturnrate);
29    }
```

Listing II.2 – Evitement d'obstacle avec Sonar

Cependant cette nouvelle décomposition dépendra de la position des capteurs sur le robot, du nombre de rayons du Laser, etc. Par conséquent, le remplacement d'un capteur par un autre impliquerait une modification du code existant car celui-ci reste écrit en fonction des données de bas niveau du Sonar. En effet, les abstractions de Player sont fournies pour certaines plateformes de robotique mais elles restent de bas niveau [45] : c'est encore au système de contrôle du robot d'interpréter les nombres retournés par les capteurs et envoyés aux effecteurs. Il faudrait encapsuler ces détails de bas niveau afin de séparer les responsabilités liées au traitement des données de celles liées à leur acquisition ou à leur exécution. Cela permettrait de faciliter le développement des applications et de pouvoir gérer les changements éventuels de matériel dans l'application.

2.2 Robot Operating System (ROS)

2.2.1 Description

L'un des middlewares les plus communément utilisé dans la communauté robotique de nos jours est ROS [10, 43]. Il supporte l'intégration de modules logiciels indépendants distribués, appelés noeuds (ROS nodes). ROS fournit un ensemble d'outils qui permettent de packager et de déployer (à travers les notions de package et de stack) les applications de robotique. Il fournit également un nombre important de librairies qui implémentent des drivers de capteurs (par exemple : Hokuyo Scanning range finder, Sharp IR range finder, etc) permettant ainsi d'encapsuler les données spécifiques et de fournir certaines fonctionnalités pour les robots (par exemple ROS navigation stack).

Les noeuds ROS sont des blocks de code et sont implémentés dans des classes C++ ou Python. Un système ROS est un graphe composé d'un ensemble de noeuds qui communiquent entre eux à travers des messages. Ces messages sont échangés avec le mécanisme *Publish/Subscribe* de manière asynchrone à travers un bus de communication appelé *ROS-Topic* [46]. Chaque topic est typpé par le type du message ROS publié dessus. La figure II.3 montre le mécanisme de communication entre 2 noeuds. En dépit de ses nombreux avantages, ROS ne gère pas l'aspect temps-réel, il délègue cet aspect à des middlewares tiers comme OROCOS [47]. Les Topics ROS permettent la communication entre ces 2 middleware.

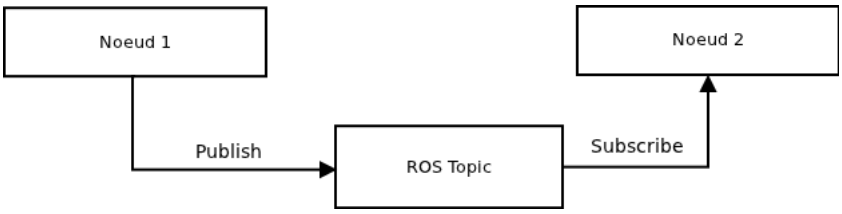

Figure II.3 – Communication entre les noeuds ROS

2.2.2 Les abstractions du matériel dans ROS

Les messages ROS [48] sont des structures de données comportant des attributs typpés. L'une des bibliothèques des messages ROS, appelée *sensor_msgs*, est dédiée aux capteurs communément utilisés. Ces messages regroupent les propriétés définies pour chaque type de capteurs. Nous citons par exemple les messages *sensor_msgs/LaserScan* pour les capteurs Laser, les messages *sensor_msgs/CameraInfo* pour les caméras, les messages *sensor_msgs/Range* pour les capteurs Infrarouge, etc. En utilisant ces messages, le code de l'application ne dépend pas d'un modèle de capteurs particulier ayant une configuration particulière mais dépend uniquement du type du capteur.
Le tableau II.2 montre les deux messages *sensor_msgs/LaserScan* et *sensor_msgs/Range* qui sont des capteurs de distance à rayons dont les propriétés diffèrent. Pour le message *LaserScan*, le *header* signifie le temps d'acquisition du premier rayon Laser alors que pour le *Range*, le *header* signifie le temps d'acquisition de la distance du rayon Infrarouge. Pour le message *LaserScan*, il est important de spécifier les attributs angles min et max de début et de fin du scan ainsi que l'attribut *angle_increment* qui indique la distance angulaire entre les mesures. Pour le message *Range*, il faut spécifier l'attribut *filed_of_view* qui représente la taille de l'arc où la distance est valide. Ensuite, dans les deux messages, on peut spécifier *min_range* et *max_range* qui définissent les valeurs minimales et maximales qui peuvent être perçues.

2.2.3 Discussion

Afin de tester les abstractions proposées par ROS, nous avons programmé un code très simple d'un composant d'évitement d'obstacles sous OROCOS qui est un framework temps-réel à base de composants C++ [47] que nous détaillerons dans la section 5. Cet exemple a été développé en fonction d'un capteur Laser en utilisant

Sensor_msgs/LaserScan.msg	Sensor_msgs/Range.msg
Header header	Header header
float32 angle_min	uint8 ULTRASOUND=0
float32 angle_max	uint8 INFRARED=1
float32 angle_increment	uint8 radiation_type
float32 time_increment	float32 field_of_view
float32 scan_time	float32 min_range
float32 range_min	float32 max_range
float32 range_max	float32 range
float32[] ranges	
float32[] intensities	

Table II.2 – Exemple de sensor messages de ROS

le topic ROS Sensor_msg : :LaserScan. Le principe est simple, si les capteurs droits du robot détectent un obstacle, le robot tourne à gauche. Si les capteurs gauches du robot détectent un obstacle, le robot tourne à droite et s'il n'y a pas d'obstacle le robot avance. Maintenant supposons que l'on veuille réutiliser ce code pour un robot disposant de 2 capteurs Infrarouges. Il faut alors adapter le code pour récupérer les informations des 2 capteurs infrarouges et faire le même traitement de données. Le listing II.3 montre les 2 codes avec Laser et Infrarouge. Le code en gris est le code qui n'a pas été modifié. Le code en bleu représente le code à modifier pour le composant écrit en fonction du capteur Laser et le code en rouge est le code qui a été modifié pour tenir compte des capteurs Infrarouges. L'impact du changement au niveau d'un composant n'est pas très important car l'exemple que nous avons pris est très simple. Imaginons maintenant une application comportant plusieurs composants qui intéragissent entre eux. Il faudrait alors faire des modifications au sein des composants et au niveau des composants avec lesquels ils intéragissent. Ces abstractions sont alors de bas niveau. Il faudrait définir des abstractions qui soient plus indépendantes des caractéristiques du matériel utilisé.

2.3 Middleware for Robots (MIRO)

2.3.1 Description

Le but de MIRO est de fournir un framework général pour le développement des applications de robotique [11]. Ce but a été défini après le développement de plusieurs applications sous différentes plateformes de robotique, avec différents langages de programmation et pour différents robots mobiles lorsque les développeurs

Capteur Laser

```
#include<sensor_msgs/LaserScan.h>
#include<geometry_msgs/Twist.h>
//include Orocos libraries
using namespace std;
using namespace RTT;
class AvoidObstaclesWithLaserScanRobot :
public RTT::TaskContext{
private :
InputPort<sensor_msgs::LaserScan>inport;

OutputPort<geometry_msgs::Twist>
public :
AvoidObstaclesWithLaserScanRobot
(const std::string& name) :
TaskContext(name),
inport("laser_in"),

outport("twist_out") {
ports()->addPort(inport);

ports()->addPort(outport);
}
 AvoidObstaclesWithLaserScanRobot() {}
};
private :
void updateHook() {
sensor_msgs::LaserScan msg;

 geometry_msgs::Twist cmd;
double midA, midB;
 if (NewData == inport.read(msg)) {



bool halt = false;
for (int i = msg.angle_min; i <= msg.angle_max; i++) {
if(msg.ranges[i] < msg.range_max) {


halt = true;
break;
}
}
 if (halt) {
midA = std::accumulate(msg.ranges.begin(),msg.ranges.end()-45, 0);
midB = std::accumulate(msg.ranges.begin()+45,msg.ranges.end(), 0);
if (midA > midB){ cmd.angular.z = -1; }
else { cmd.angular.z = 1; }
}
else {
cmd.linear.x = 1;
outport.write(cmd);}
} }
ORO_CREATE_COMPONENT(AvoidObstaclesWithLaserScanRobot)
```

Capteurs Infrarouge

```
#include <sensor_msgs/Range.h>
#include <geometry_msgs/Twist.h>
//include Orocos libraries
using namespace std;
using namespace RTT;
class AvoidObstaclesWithIR :
public RTT::TaskContext{
private :
InputPort<sensor_msgs::Range>inport_infrared_left;
InputPort<sensor_msgs::Range>inport_infrared_rightt;
OutputPort<geometry_msgs::Twist>
public :
AvoidObstaclesWithIRRobot
(const std::string& name) :
TaskContext(name),
inport_infrared_left("infrared_left_in"),
inport_infrared_right("infrared_right_in"),
outport("twist_out") {
ports()->addPort(inport_infrared_left);
ports()->addPort(inport_infrared_right);
ports()->addPort(outport);
}
 AvoidObstaclesWithIRRobot() {}
};
private :
void updateHook() {
sensor_msgs::Range msg_infrared_left;
sensor_msgs::Range msg_infrared_right;
 geometry_msgs::Twist cmd;
double ir_left = -1.0, ir_right = -1.0;
if (NewData == inport_infrared_left.read(msg_infrared_left))
ir_left = msg_infrared_left.range;
if(NewData == inport_infrared_right.read(msg_infrared_right))
ir_right = msg_infrared_right.range;
bool halt = false;
if ( ((ir_left > msg_infrared_left.min_range)
&& (ir_left < msg_infrared_left.max_range)) ||
((ir_right > msg_infrared_right.min_range)
&& (ir_right < msg_infrared_right.max_range)) ) {
halt = true;
}


 if (halt) {


if(ir_left > ir_right) { cmd.angular.z = -1; }
else { cmd.angular.z = 1; }
}
else {
cmd.linear.x = 1;
outport.write(cmd);
} }
ORO_CREATE_COMPONENT(AvoidObstaclesWithIR)
```

Table II.3 – à gauche le code écrit en fonction du capteur Laser, à droite le code après sa modification avec les Infrarouges

ont réalisé l'importance de la définition d'un framework général pour éviter de réimplémenter les mêmes applications si on change de robot. MIRO a été conçu et implémenté en utilisant l'approche orientée objet ainsi que l'architecture du middleware CORBA (*Common Object Request Broker Architecture*) [49] qui permet d'assurer la communication entre les différents modules du robot et entre plusieurs robots.

L'architecture de MIRO est composée de trois couches :

- Une couche dispositif (*device layer*).
- Une couche service (*service layer*).
- La classe framework MIRO (*MIRO framework class*) qui fournit un ensemble

de modules fonctionnels fréquemment utilisés en robotique comme la localisation, la plannification de chemin, etc.

2.3.2 Les abstractions du matériel dans MIRO

Le framework MIRO contient trois couches dont deux qui permettent d'abstraire le matériel utilisé.
La couche dispositif encapsule les messages de communication de bas niveau (liés au bus de communication, etc.) dans des appels de méthodes pour l'invocation des services demandés. Cette couche fournit une interface pour les capteurs et les actionneurs d'un robot. Ces derniers sont définis comme des objets qui peuvent être interrogés et controlés par leurs méthodes. La couche dispositif est dépendante des détails des robots et de la plateforme.
La couche service fournit des abstractions de services pour les capteurs et les actionneurs à travers l'interface de définition CORBA (*CORBA Interface Description Language : IDL*) et implémente ensuite ces services d'une manière indépendante des plateformes. L'interface `RangeSensor_IDL` définie dans le langage IDL permet d'abstraire les données de Laser et de l'infrarouge II.3.

```
1   interface RangeSensor
2   {
3      /**
4       * The specific sensor is either masked out or didn't provide usefull data.
5       */
6      const long INVALID_RANGE = -2;
7      /**
8       * The scan value is bigger than the maximum distance the range
9       * sensor can measure.
10      */
11     const long HORIZON_RANGE = -1;
12     //! The range sensor does not send events
13     const long NONE_PUSHING = 0;
14     //! The range sensor sends a full scan with its events (@ref RangeScanEventIDL).
15     const long FULL = 1;
16     //! The range sensor sends one sensor group scan with its events (@ref RangeGroupEventIDL).
17     const long GROUPWISE = 2;
18     //! The range sensor sends a bunch of sensor readings with its events (@ref RangeBunchEventIDL).
19     const long BUNCHWISE = 3;
20     //! Query the layout of a range sensor type.
21     ScanDescriptionIDL getScanDescription();
22     //! Query a range sensor group.
23     RangeGroupEventIDL getGroup(in unsigned long id) raises(EOutOfBounds);
24     //! Wait and query a range sensor group.
25     RangeGroupEventIDL getWaitGroup(in unsigned long id) raises(EOutOfBounds, ETimeOut);
26     //! Query a range sensor.
27     RangeScanEventIDL getFullScan();
28     //! Wait and query a range sensor.
29     RangeScanEventIDL getWaitFullScan() raises(ETimeOut);
30  };
```

Listing II.3 – Interface RangeSensor_IDL de MIRO

2.3.3 Discussion

Le listing II.4 montre un exemple présenté dans les travaux de Krüger et al. [50] programmé en Python en utilisant les interfaces fournies par MIRO. Dans cet exemple, le robot est équipé par des capteurs sonar. Il avance jusqu'à ce que l'obstacle le plus proche détecté soit plus proche que 3000 millimètres, dans ce cas il s'arrête.

```
1 import pyMiro
2 sonar=pyMiro.getSonar()
3 loko=pyMiro.getMotion()
4 while(1):
5 scan=sonar.getFullScan()
6 front_scan=scan.range[0]
7 if min(front_scan)<3000:
8 loko.limp()
9 else:
10 loko.setLRVelocity(50,50)
```

Listing II.4 – Evitement d'obstacle avec les abstractions de MIRO

L'abstraction *getFullScan()* (ligne 5) est définie dans l'interface `RangeSensor` présentée dans la section précédente. L'abstraction *limp()* est définie dans une interface appelée `Motion` afin d'abstraire les mouvements des effecteurs, elle représente un arrêt passif (c.-à-d. le robot ne s'arrête pas complètement).
Le retour d'expérience de Krüger et al. dans [50] sur l'utilisation de MIRO montre qu'on peut intégrer des nouveaux capteurs à MIRO en implémentant les interfaces fournies mais n'évoque pas le changement de capteur dans les applications existantes.
Supposons maintenant que l'on veuille réutiliser ce même code mais avec un capteur Laser. On peut remplacer *getSonar()* (ligne 2) par *getLaser()*, scan.rang[0] (ligne 6) par l'identifiant du rayon Laser placé à l'avant du robot et le code fonctionnerait. Cependant, il faut tenir compte de la portée du rayon Laser, de sa position s'il est placé à gauche du centre du robot par exemple, etc. On peut dire que les abstractions de MIRO permettent une intégration de nouveaux capteurs en supposant qu'il remplit toutes les hypothèses sur les besoins du code. Cette intégration implique l'adaptation du code existant et donc des changements qui peuvent être assez importants.
En revanche, il faudrait modifier les paramètres spécifiques à la portée du capteur par exemple. Typiquement dans ce cas, 3000 mm est la distance à partir de laquelle le robot doit s'arrêter. Si la portée du nouveau capteur est inférieure à 3000 millimètres, il faut modifier ce paramètre. Plus les tests sont importants, plus les paramètres à modifier le seront aussi. Ces changements ont un impact sur la visibilité du code car les détails de bas niveau seront confondus avec les détails d'implémentation et ceux

de l'algorithme. Si on prend maintenant un autre exemple où on voudrait utiliser des capteurs tactiles, il faut réécrire tout le code et implémenter les abstractions d'une nouvelle interface.
Il faudrait alors encapsuler toutes les données de bas niveau à travers des abstractions de plus haut niveau sous forme de requêtes sur l'environnement par exemple.

2.4 Python Robotics (PyRo)

2.4.1 Description

PyRo [12, 44] est un environnement de développement de robotique en Python qui permet aux étudiants et aux chercheurs d'explorer différents thèmes en robotique indépendamment des robots utilisés. Il a été intégré à plusieurs cours académiques car il offre un support simple pour les étudiants pour le développement des applications de robotique.
Le but de PyRo est de faciliter le développement des applications de robotique et d'assurer leur portabilité, sans la modification du code, à travers des abstractions de haut niveau qui encapsulent les détails de bas niveau.
PyRo est composé d'un ensemble de classes Python qui encapsulent les détails de bas niveau. Les utilisateurs programment leurs applications en utilisant une API. Cette dernière est implémentée avec une hiérarchie orientée objet offrant ainsi une couche d'abstraction.

2.4.2 Les abstractions du matériel dans PyRo

Les abstractions du matériel définies dans PyRo sont les suivantes :

- *Range sensors* : Indépendamment du capteur utilisé, cette interface définit des abstractions pour les capteurs Laser, Infratouge et Sonar.
- *Robot units* : L'unité de la distance retournée par les capteurs peut être en mètres ou en millimètres ou simplement une valeur numérique où les grandes valeurs indiquent un espace libre et les petites valeurs indiquent la présence d'un obstacle proche. Robot unit est une nouvelle unité de mesure qui unifie toutes les autres.
- *Sensor groups* regroupe les capteurs selon leur emplacement. L'utilisateur n'aura plus à se soucier du nombre de capteur disponibles mais utilisera sim-

plement des abstractions du type : *front, frontleft, left, etc.* comme le montre la figure II.4. Les abstractions proposées sont les suivantes : *front-all* pour tous les capteurs avant du robot, *front* pour les capteurs placés à l'avant du robot, *front-left* pour les capteurs à l'avant gauche, *front-right* pour les capteurs placés à l'avant droit du robot. De la même façon, les abstractions *back*, *back-right* et *back-left* sont définies par rapport à l'arrière du robot.

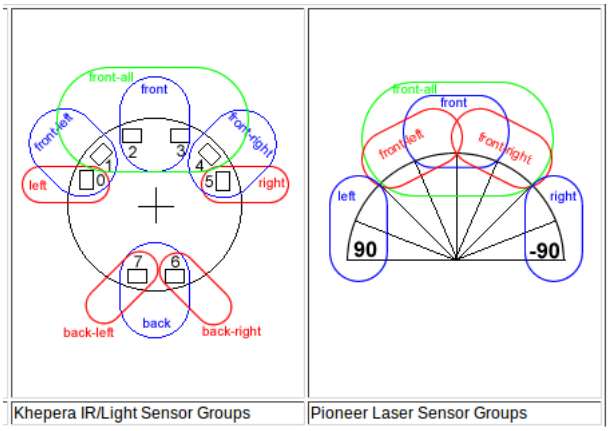

Figure II.4 – Le regroupement des capteurs dans PyRO

- *Motion control* : définit des abstractions pour les commandes des effecteurs : `move(translate, rotate)` et `motors(leftpower, rightpower)`
- *Devices* : Permet d'introduire des nouveaux dispositifs qui n'ont pas encore été pris en charge.

2.4.3 Discussion

PyRo propose des abstractions pour le matériel sous forme d'actions de plus haut niveau que les commandes spécifiques envoyées au robot et de regroupement de capteurs selon leurs positions. Prenons maintenant un exemple défini en fonction de ces abstractions (voir listing II.5). Cet exemple représente une tâche d'évitement d'obstacle très simple, si un obstacle est rencontré à gauche le robot tourne à droite, si un obstacle est rencontré à droite le robot tourne à gauche et s'il n'y a pas d'obstacle, le robot avance.

```
1 # if approaching an obstacle on the left side
2 # turn right
3 # else if approaching an obstacle on the right side
4 # turn left
```

```
5  # else go forward
6  from pyro.brain import Brain
7  class Avoid(Brain):
8  def step(self):
9  safeDistance = 1 # in Robot Units
10 #if approaching an obstacle on the left side, turn right
11 if min(self.get('robot/range/front-left/value')) < safeDistance:
12         self.robot.move(0,-0.3)
13 #else if approaching an obstacle on the right side, turn left
14 elif min(self.get('robot/range/front-right/value')) < safeDistance:
15         self.robot.move(0,0.3)
16 #else go forward
17 else:
18         robot.move(0.5, 0)
```

Listing II.5 – Evitement d'obstacle programmé avec PyRo extrait de [12]

Le code présenté est très simple grâce aux abstractions proposées, il nécessite des connaissances sur la vitesse du robot (ligne 13), son angle de rotation (ligne 10) et sur les valeurs retournées par les capteurs. La fonction *min* (lignes 6 et 9) est une fonction Python permettant de retourner le minimum des valeurs lues à partir des capteurs. Ce code peut être utilisé pour tout robot équipé de capteurs à rayons avant gauche et avant droit.
Supposons maintenant que l'on veuille réutiliser ce même code mais avec des capteurs tactiles, il faut modifier alors les conditions sur les valeurs des capteurs et sur leurs positions. Par conséquent, bien que ces abstractions soient de plus haut niveau que les capteurs en permettant de les regrouper et de ne pas se soucier du nombre de rayons, elles restent insuffisantes. En effet, elles ne permettent pas de fournir des informations de haut niveau sur l'environnement car elles sont spécifiques à la position des capteurs et ne font que des regroupements de ces derniers. Il faudrait définir des abstractions de plus haut niveau afin de garantir une indépendance par rapport à la position des capteurs en fournissant des informations sur l'environnement du robot.

3 Conclusion

Une étude complète des middleware de robotique existants est présentée dans [21] montrant les particularités de chaque middleware et les avantages de son utili-

sation. Dans ce chapitre, nous avons présenté certains des middleware de robotique qui traitent l'aspect abstraction du matériel. Nous avons vu que ces abstractions sont insuffisantes car elles restent de bas niveau et n'isolent pas bien de certains changements sur les capteurs. En effet, le traitement des données des capteurs reste confondu avec les détails d'implantation de l'application. Ces abstractions ne permettent pas d'encapsuler les détails de bas niveau et de faciliter le développement des applications de robotique. De plus, les middleware ne traitent pas les aspects qui concernent la variabilité algorithmique.

Afin de faciliter le développement de ces applications, il faudrait permettre aux roboticiens de manipuler des concepts qu'ils ont l'habitude d'utiliser pour le développement de leurs applications. Ces concepts peuvent être des concepts de haut niveau du domaine comme ceux proposés par les DSML (langage de modélisation spécifiques au domaine). Cependant, les concepts proposés par les DSML manquent souvent de sémantique opérationnelle. Il faudrait alors définir des abstractions de haut niveau qui permettent d'obtenir des informations sur l'environnement du robot et de définir des actions de haut niveau d'une part et d'encapsuler les détails algorithmiques d'autre part. ne sont autres que les concepts du domaine. Les techniques de l'ingénierie dirigiée par les modèles ont été appliquées dans ce contexte afin de définir ce qui est appelé les DSML (langage de modélisation spécifique). Dans le chapitre suivant, nous présentons les différentes phases de création d'un DSML. Nous présenterons ensuite les DSML pour la robotique existants.

Langages de modélisations spécifiques à la robotique

1 Introduction

La définition des applications de robotique est souvent considérée comme un processus complexe car ceci requiert une expertise du domaine (matériel utilisé, tâche à programmer, etc.) ainsi qu'une expertise en développement (plateforme, architecture, etc.). Nous avons présenté dans le premier chapitre la variabilité dans la robotique mobile. Les middleware ont essayé de pallier le problème de la variabilité du matériel en introduisant des abstractions qui restent toutefois de bas niveau comme nous l'avons vu dans le chapitre précédent. D'autre part, chaque middleware s'appuie sur un protocole de communication qui lui est propre ou qui s'appuie sur le standard CORBA. Par conséquent, un problème de portabilité d'un middleware vers un autre a été constaté en plus de la difficulté de faire interopérer les applications entre elles.
L'une des priorités pour la communauté robotique consiste alors à définir des abstractions communes pour garantir l'indépendance des applications des middleware, assurer leur portabilité et ainsi optimiser le processus de développement.
Afin de répondre à cet objectif, il faudrait tout d'abord réduire le fossé sémantique entre les développeurs et les roboticiens en utilisant un langage commun compréhensible par les roboticiens d'un côté et par les développeurs d'un autre côté.
Ce langage commun est appelé langage spécifique à un domaine ou langage dédié (DSL : domain specific language), il doit répondre au besoin de définir des spécialisations du domaine étudié [51].
Les langages de modélisation spécifiques au domaine (DSML : Domain Specific Mo-

deling Language) sont des DSL basés sur les modèles qui visent à définir des abstractions de haut niveau et à accélérer le processus du développement à travers l'utilisation des modèles. Ils permettent une meilleure compréhension des systèmes représentés ainsi qu'une accessibilité au développement des applications pour les roboticiens à travers des notations appropriées du domaine.
Dans ce chapitre, nous commençons par une présentation générale des DSML et des techniques d'ingénierie dirigée pas les modèles. Nous présentons ensuite le cycle de vie d'un DSML puis les DSML existants pour la robotique.

2 Langages de Modélisation Spécifiques aux domaines (DSML)

Un DSL est un *langage de programmation ou un langage de spécification qui offre, à travers des notations appropriées et des abstractions, une puissance expressive axée sur, et généralement limitée à, un domaine particulier* [14]. Les DSL sont souvent des langages déclaratifs, ils peuvent par conséquent être considérés comme des langages de spécification aussi bien que des langages de programmation [14]. Le développement d'un DSL requiert une expertise du domaine auquel il est dédié ainsi qu'une expertise en développement des langages [52]. L'un des objectifs principaux d'un DSL consiste à réduire l'ambiguïté de la terminologie et le fossé sémantique entre les experts du domaine d'application et les développeurs et ainsi convenir d'un **ensemble d'abstractions communes pour un domaine particulier**. Par conséquent, les experts du domaine doivent être mis au centre du développement du DSL [53].

Idéalement, un DSL apporterait plusieurs avantages aux développeurs ainsi qu'aux experts du domaine auquel il est dédié. Plusieurs travaux soulignent ces avantages, notamment Kieburtz et al. [54] qui ont réalisé une comparaison entre deux approches dans le cadre d'une étude empirique. La première consistait au développement d'une application en utilisant un ensemble de templates fournissant des fonctionnalités et des types génériques et la deuxième était l'utilisation d'un générateur de templates à partir d'un DSL. Ils ont constaté que l'utilisation de la deuxième approche améliorait considérablement la **productivité** des développeurs, la **fiabilité** de l'application et garantissait une **facilité d'utilisation** offrant ainsi un **outil de réflexion et de communication** ([55], page 41) aussi bien pour les développeurs que pour les ex-

perts du domaine. Grâce à cette facilité d'utilisation, les experts du domaine peuvent comprendre les programmes définis avec le DSL, les modifier et même les développer eux-même [14]. D'après Van Deursen [56], l'utilisation des DSLs renforcerait également la **maintenance** des applications et assurerait leur **portabilité**. Idéalement, un DSL devrait être **réutilisable** [57] dans le sens où l'on peut réutiliser les connaissances capitalisées du domaine afin de définir plusieurs programmes différents. Cette réutilisabilité découle également de **l'indépendance des DSL des plateformes cibles** ([55], page 43) car lors de la conception d'un DSL, une séparation des reponsabilités est réalisée entre les notations du domaine et les détails d'implémentations liés aux plateformes.
En dépit de tous ces avantages, les DSLs présentent certaines limites comme par exemple, un coût non négligeable de conception, d'implémentation et d'apprentissage pour leur utilisation [14]. Un DSL bien défini apporterait alors des solutions aux limites constatées.

Les DSL sont souvent définis comme des langages de modélisation spécifiques au domaine (DSML) qui incluent une infrastructure pour des transformations automatiques à partir de modèles vers un code source et des artefacts [2]. Les techniques de l'ingénierie dirigée par les modèles permettent la conception des DSML. Dans la section suivante nous présentons ces techniques.

2.1 Les techniques d'ingénierie dirigée par les modèles

L'Ingénierie Dirigée par les Modèles (IDM) ou Model Driven Engineering (MDE) [15] répond au besoin de définir des abstractions du domaine en plaçant les modèles au centre du développement logiciel.
Un modèle est *une représentation simplifiée d'un système destiné à améliorer notre capacité à comprendre, prévoir et éventuellement contrôler le comportement de ce système* [58]. Nous pouvons dire, par exemple, qu'une carte géographique est un modèle. L'interprétation de cette carte diffère d'une personne à une autre en l'absence de légende. Nous devons donc définir une légende standard pour cette carte afin de pouvoir l'interpréter de manière unique. C'est de cette même façon que nous pouvons définir le lien entre les modèles et les méta-modèles. Un méta-modéle (niveau M2 de la figure III.5) est un langage dont l'interprétation est unique car il définit les concepts d'un domaine particulier et il est lui-même conforme à un méta-méta-

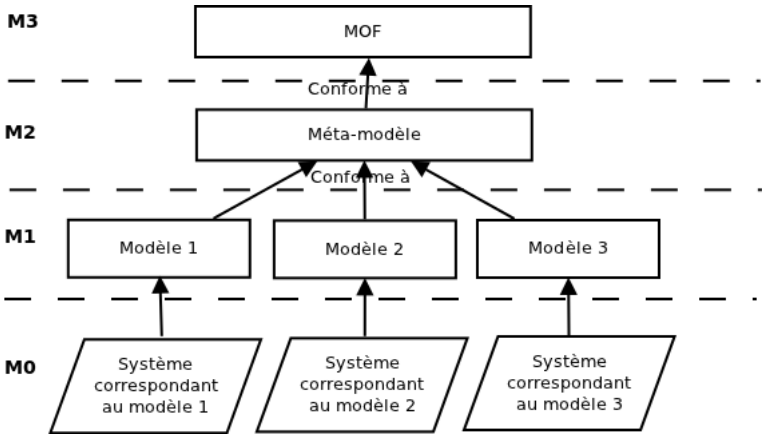

Figure III.5 – Les différents niveaux de modélisation

modèle (MOF : Meta Object Facility) (niveau M3 de la figure III.5). Les modèles (niveau M1 de la figure III.5), instances du méta-modèle, respectent donc les notations du domaine et contiennent une combinaison particulière de ses concepts.
L'OMG (Object Management Group) est un consortium d'industriels et de chercheurs dont l'objectif est d'établir des standards permettant de résoudre les problèmes d'interopérabilité des systèmes d'infomation [59]. L'architecture proposée par l'OMG pour les différents niveaux de modélisation est présentée dans la figure III.5.

La MDE est une généralisation de l'approche MDA (Model Driven Architecture), proposée et soutenue par l'OMG. La MDA applique la séparation des préoccupations qui consiste à élaborer des modèles métier (indépendants des détails techniques des plateformes) et des modèles spécifiques aux plateformes (pour la partie technique d'une plateforme). L'avantage le plus important qu'offre MDA est la pérennité des modèles indépendants des plateformes grâce à la modélisation des spécifications métier. Dans ce qui suit nous présentons les différents niveaux de modélisation en MDA et les transformations entre ces modèles ainsi que les outils dédiés à la modélisation et aux transformations.

2.1.1 Les niveaux de modélisation

Trois niveaux de modélisation interviennent en MDA :

– Les Modèles Indépendants des Plateformes (PIM : Platform Independent Model). Ces modèles sont proches des contraintes et des considérations des experts du domaine et permettent de représenter la structure et les opérations

du système et ce, d'un point de vue indépendant des plateformes.
- Les Modèle Spécifiques aux Plateformes (PSM : Platform Specific Model). Ces modèles résultent d'une association du PIM aux détails techniques d'une plateforme donnée. À partir du PSM le code vers la plateforme cible est généré.
- Les Modèles de Description des Plateformes (PDM : Platform Description Model). Ces modèles permettent de spécifier la façon dont les fonctionnalités de la plateforme sont implémentées. Ils spécifient également comment ces fonctionnalités sont utilisées.

Un PIM est conforme à un méta-modèle indépendant des plateformes et un PSM est conforme à un méta-modèle représentant les concepts d'une plateforme spécifique. Nous allons maintenant présenter les différents outils permettant de réaliser les différents niveaux de modélisation notamment les méta-modèles et les modèles.

2.1.1.1 Outils Le projet de modélisation d'Eclipse (EMP) [60] offre une multitude d'outils dédiés au développement dirigé par les modèles. Une partie de l'EMP comprend EMF (Eclipse Modeling Framework) qui permet la définition de méta-modèles sous forme de fichiers Ecore. Une instance d'un méta-modèle est un modèle défini sous forme de fichier XMI.
Papyrus [61] est un outil d'édition graphique de modèles, basé sur l'environnement Eclipse EMF, pour UML2 [62]. Conformément à son objectif principal, il met en œuvre la spécification standard complète de UML2 notamment les diagrammes de structure (c.-à-d. diagramme de classe, diagramme de déploiement, etc.), les diagrammes de comportement (c.-à-d. le diagramme d'activités, diagramme de machine à états) et les diagrammes d'intéraction (c.-à-d. diagramme de séquence, etc.). Papyrus est également conforme au standard graphique DI (Diagram Interchange) qui permet d'échanger les données graphiques. Papyrus fournit aussi un support large pour les profils UML [63]. Ces derniers permettent d'étendre UML afin de l'adapter à plusieurs domaines d'applications.

Dans le cadre du projet PROTEUS, nous avons utilisé Papyrus afin de définir un éditeur graphique permettant de modéliser des scénarios de robotique en utilisant les concepts du domaine.
Afin d'avoir une application exécutable vers une plateforme cible à partir des différents modèles présentés précédemment, des transformations sont définies pour transformer les modèles en code exécutable. Nous les présentons dans la section suivante.

2.1.2 Les transformations

MDA passe par la transformation respective de ses modèles afin d'obtenir une application exécutable à partir de ces derniers. Ces transformations sont effectuées :

- du PIM vers le PSM (de modèle vers modèle) : pour établir les liens entre les concepts généraux et les concepts spécifiques aux plateformes ;
- du PSM vers du code (de modèle vers texte) : pour obtenir du code exécutable correspondant au modèle spécifique à la plateforme d'exécution.

Théoriquement, il faudrait que le PDM intervienne dans les transformations de PIM vers PSM pour spécifier les fonctionnalités relatives à la plateforme utilisée. Cependant, pour la définition des transformations, il faudrait au préalable définir des méta-modèles du PIM, du PDM et du PSM. La définition des méta-modèles du PIM et du PSM ne pose pas de problème particulier. En revanche, le méta-modèle du PDM n'est actuellement pas réalisable car il devrait permettre de construire des modèles de toutes les plateformes et vu que ces dernières sont différentes, il n'existe pas de nos jours de solution à ce problème. Les transformations existantes se résument alors à des transformations du PIM vers le PSM (de modèle vers modèle) et du PSM vers le code (de modèle vers texte).
Plusieurs outils sont proposés pour réaliser ces transformations. Nous les présentons dans la section suivante.

2.1.2.1 Outils Les outils disponibles aujourd'hui basés sur les techniques d'ingénierie dirigée par les modèles permettent de construire des méta-modèles, des modèles et leurs équivalents en XMI (XML Metadata Interchange) [64] ou en UML facilement manipulables par les outils de génération de code assurant ainsi la génération de code vers multiples et diverses plateformes cibles.
Une **plateforme cible** est une plateforme constituée de ressources logicielles et matérielles servant de support d'exécution à l'application générée. Nous soulignons la différence entre un outil de génération de code et un générateur de code.
Un **outil de génération de code** est un compilateur qui permet de définir des règles de transformations entre le DSL et la plateforme cible tandis qu'un **générateur de code** est un programme qui établit des règles de transformations entre le DSL et la plateforme cible. Le générateur de code est implémenté avec un outil de génération de code.
Il existe plusieurs outils de génération de code, certains de modèle vers modèle

comme Epsilon [65], Kermeta [66] et d'autres, de modèle vers texte, comme Acceleo [67] et TOM-EMF [68].
Acceleo permet de définir des transformations de Modèle vers Texte (approche MDA) et supporte les standards de modélisation comme UML, XMI, etc.. La syntaxe mise en oeuvre par Acceleo se base sur des templates qui contiennent des informations tirées sur les modèles en entrée (à travers des requêtes) et leur associent le code à générer à travers des conditions et des boucles.
Dans le cadre de cette thèse et du projet ANR PROTEUS, nous utiliserons Acceleo comme outil de génération de code.
Dans la section suivante, nous présentons les différentes étapes de conception d'un DSML.

3 Cycle de vie d'un DSML

Avant de concevoir un DSML, plusieurs questions doivent être soulevées :

- Par qui sera conçu le DSML ?
- Pour qui sera conçu le DSML ?
- Quels sont les concepts du domaine qui doivent être représentés et comment les représenter ?

Ces questions permettent d'orienter les choix de conception d'un DSML afin de réduire le coût en terme de maintenance et de développement et garantir ainsi des solutions aux problèmes existants.
La création d'un DSML est un processus itératif. Un DSML se base souvent sur une compréhension limitée et simple du domaine, puis évolue progressivement avec l'intervention des experts du domaine. Nous pouvons distinguer sans ambiguïté quatre phases qui déterminent le cycle de vie d'un DSL. Tout d'abord, une phase d'analyse du domaine afin d'identifier les abstractions appropriées. Une abstraction est un concept. Ce concept fait partie du domaine modélisé et représente une entité abstraite ou concrète dans ce domaine [69]. L'analyse du domaine est succédée d'une phase de conception afin de représenter ces abstractions puis d'une phase d'intégration des plateformes. La dernière phase est enfin l'utilisation du DSL [14].

3.1 Analyse du domaine

Les deux premières questions présentées précédemment doivent être traitées dans la phase de l'analyse du domaine. L'étape de l'analyse du domaine consiste à rassembler les connaissances du domaine et à les regrouper en notions sémantiques et en opérations. La modélisation des domaines, désignée par l'ingénierie des domaines (Domain engineering) [70], est un aspect de l'ingénierie logicielle introduit afin de pallier les problèmes de réutilisabilité. L'ingénierie des domaines peut être utilisée afin de construire des librairies réutilisables pour les DSLs [14].
Selon l'utilisateur final de l'application, l'identification des concepts du domaine peut varier. Prenons par exemple le cas de la robotique mobile. Pour un expert en robotique, les concepts relevant du contrôle, de la communication, de la perception, de l'action et de l'environnment réel du robot ou de la simulation définissent les concepts de base de la robotique mobile.
Un développeur, quant à lui, traitera les aspects techniques liés à la programmation et à l'architecture logicielle des plateformes de robotique (langages à base de composants, langages orientés objets, etc.).
La délimitation du domaine est un processus qui pourrait être conflictuel s'il n'est pas clairement défini. Simos et al. [71] ont distingué deux catégories d'utilisation du domaine :

- Le domaine comme étant un "monde réel" : Ceci désigne le domaine du monde où le travail final va être réalisé. On retrouve cette vision du domaine en intelligence artificielle, en génie cognitif ou encore en programmation orientée objet (OO). L'analyse du domaine en OO traduit souvent l'analyse du domaine en une description orientée-objet des entités et des transactions dans le monde réel. Cette description forme une base pour l'implantation d'un système qui supporte ces activités du monde réel. En d'autres termes, cette théorie se base souvent sur le principe que les experts d'un domaine ne prêtent pas beaucoup d'importance à l'aspect logiciel d'une application.
- Le domaine comme étant un "ensemble de systèmes" : Cette définition du domaine se base sur la théorie de l'analyse du domaine orientée par la réutilisabilité. Par opposition au domaine en tant que monde réel, les systèmes sont eux même l'objet de l'étude du domaine dans cette catégorie. Le domaine est défini comme une famille ou un ensemble de systèmes incluant des fonctionnalités communes dans un domaine particulier. La modélisation des similarités

et des variabilités à travers les systèmes existants dans le domaine est la base de cette définition du domaine.

Dans certains cas, les concepts d'un DSL en général et d'un DSML en particulier peuvent être extraits à partir d'un système existant et les abstractions du domaine peuvent être dérivées directement à partir de ce système existant. Si l'architecture du système existant est documentée en utilisant un langage de modélisation graphique par exemple, cette architecture peut être une source valuable pour l'obtention des abstractions du domaine [2].

Parmi les approches de modélisation des domaines, nous citons l'analyse de domaine orientée par les features (FODA : Feature Oriented Domain Analysis) [72] et les ontologies [73].

3.1.1 Les ontologies

L'ontologie est une approche de modélisation des domaines qui vise à fournir une compréhension des éléments d'un domaine particulier. Elle permet de représenter des bases de connaissances qui offrent un contenu sémantique réutilisable et accessible depuis diverses applications. Elle définit des concepts ainsi que leur signification et les relations entre eux pour représenter un domaine de connaissances. Une ontologie doit être exprimée suivant une syntaxe donnée, associée à une interprétation donnée. Le langage OWL (Ontology Web Language) est l'un des langages qui permettent de représenter les ontologies. Nous le présentons dans ce qui suit.

3.1.1.1 Le langage OWL (Ontology Web Language) Le langage OWL est un langage pour la représentation des ontologies Web. Il est compatible avec le Web sémantique ce qui permet aux utilisateurs de donner une définition formelle aux termes qu'ils créent. Ainsi les machines peuvent raisonner sur ces termes. La terminologie de OWL se base essentiellement sur espaces de nommages (namespace), les classes et les propriétés.

Un namespace est un conteneur qui fournit le contexte du contenu d'un fichier OWL. Une classe encapsule la signification d'un concept. Des hiérarchies de classes peuvent être créées en déclarant qu'une classe peut être une sous-classe (*subClassOf*) d'une autre classe. Une propriété décrit une association entre les classes. Une hiérarchie de propriétés peut également être créée en précisant qu'une propriété est une sous propriété d'une autre propriété (*Property :IsA*). Une composition générique des pro-

priétés est exprimée avec le mode clé *Property :hasA*, cela veut dire qu'une propriété peut s'appliquer à d'autres propriétés.
Dans le cadre du projet ANR PROTEUS qui a défini le contexte de cette thèse, nous nous baserons sur une ontologie existante du domaine de la robotique mobile, développée dans le cadre de ce projet, afin d'extraire les abstractions du domaine appropriées pour notre DSML : RobotML. L'ontologie de RobotML sera présentée dans la section 3 du chapitre 5.

3.2 Conception des DSML

Dans cette section, nous tentons de répondre à la troisième question sur l'extraction des abstractions pertinentes du domaine et leur représentation. La conception d'un DSML doit permettre une définition non ambigûe d'abstractions de haut niveau représentant les concepts identifiés à partir de l'analyse du domaine. Ces abstractions doivent être indépendants des détails des plateformes (middleware) et doivent représenter les concepts du domaine. La conception d'un DSL passe par deux étapes importantes qui sont la définition de la syntaxe abstraite et la définition de la syntaxe concrète ([55], p.175). À ces deux étapes peut s'ajouter la définition de la sémantique de la syntaxe abstraite. Dans cette section nous présentons ces différentes étapes.

3.2.1 Définition de la syntaxe abstraite

La syntaxe abstraite d'un DSML peut être définie de deux façons principales qui peuvent être complémentaires. La première façon consiste à définir le modèle du domaine sous forme d'un méta-modèle représentant les abstractions du domaine et les relations entre elles. La deuxième façon consiste à étendre les concepts d'UML sous forme d'un profil afin de les adapter aux abstractions identifiées dans le méta-modèle. Dans cette section nous présentons donc le modèle de domaine en tant que méta-modèle ainsi que les profils UML.

3.2.1.1 Modèle du domaine : Méta-modèle La syntaxe abstraite appelée aussi modèle du domaine regroupe les concepts du domaine cible du DSL, leurs propriétés et les relations entre eux dans un modèle du domaine. Le modèle du domaine formalise les connaissances du domaine et doit être validé par les experts du domaine.
Les éléments du DSL peuvent aussi avoir une sémantique. La définition de cette

dernière n'est pas une étape indispensable pour la conception d'un DSL. La sémantique peut être statique ou dynamique. Dans le premier cas, elle permet de définir la sémantique qui ne peut pas être exprimée directement au niveau du modèle du domaine comme par exemple les éléments invariants et les relation entre eux. Quant à la sémantique dynamique, elle permet de définir les effets comportementaux résultant de l'utilisation des éléments du DSL. Elle définit également la façon dont les éléments du DSL doivent intéragir à l'exécution. Le comportement peut être exprimé de différentes façons allant de modèles de flots de contrôle de haut niveau en passant par des modèles de comportements plus détaillés jusqu'à une spécification textuelle précise.
Dans le cas des DSML, le modèle du domaine est défini comme un PIM qui représente les concepts du domaine. Des modèles spécifiques aux plateformes peuvent aussi être définis afin de représenter les concepts des plateformes cibles du DSML. Ces modèles, si définis, interviendront dans l'étape intégration des plateformes des DSML.

3.2.1.2 Profil UML Un profil étend les concepts d'UML afin de les adapter à un domaine d'application. L'extension d'UML à travers les profils passe par trois concepts principaux [63] :

- *Stereotype* : Un stéréotype étend une méta-classe existante du méta-modèle UML pour en changer la sémantique. Un stéréotype peut posséder des propriétés (*tagged value*) et être sujet à des contraintes. Un stéréotype est une extension limitée d'une méta-classe, on ne peut pas définir un stéréotype sans définir le lien vers la méta-classe qu'il étend. UML permet egalement d'associer à chaque stéréotype sa propre notation sous forme d'une icône et/ou un cadre (shape) pour le distinguer graphiquement du concept qu'il étend.
- *Tagged Value* : Ce sont des propriétés des stéréotypes qui offrent un moyen de spécifier les caractéristiques du concept UML étendu. Une propriété a un type qui appartient impérativement à UML sinon il doit être défini et annexé au profil.
- *Constraint* On peut définir des contraintes sur le méta-modèle UML afin de le spécialiser. Ces contraintes sont définies avec le langage OCL (Object Constraint Language).

Le profil UML sert de support pour la définition d'un éditeur graphique qui va utiliser les différentes icônes associées aux stéréotypes par exemple. Il est également possible

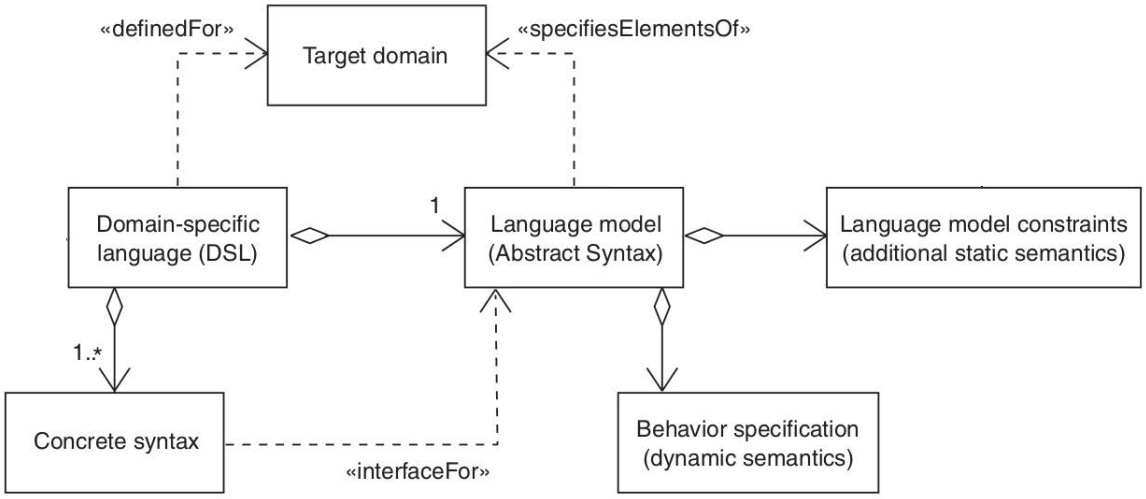

Figure III.6 – Les artefacts du DSL - extrait de [2]

de vérifier les contraintes sur l'association et la composition des différents éléments proposés à travers cet éditeur et de valider ainsi les modèles réalisés. Dans la section suivante, nous précisions en quoi consiste la définition de la syntaxe concrète d'un DSML.

3.2.2 Définition de la syntaxe concrète

La syntaxe concrète est une interface à travers laquelle sont représentées les abstractions définies dans la syntaxe abstraite [2]. Chaque DSL peut avoir plusieurs syntaxes concrètes : graphique et textuelle. Pour un DSML, la syntaxe concrète est présentée sous forme graphique. Elle est représentée à travers des outils permettant à l'utilisateur de créer des modèles qui représentent une combinaison particulière des éléments de la syntaxe abstraite. La définition de la syntaxe concrète est une étape importante du point de vue de l'utilisateur final du DSML.

La figure III.6 résume les différents artefacts intervenant dans la conception d'un DSL. Un DSL correspond à un domaine, il possède une syntaxe abstraite qui peut avoir une sémantique statique ou (et) dynamique (que nous ne détaillerons pas car nous ne traitons pas ces parties dans cette thèse). Un DSL a une ou plusieurs syntaxes concrètes qui correspondent à cette syntaxe abstraite. Dans le cadre du projet PROTEUS, Papyrus a été utilisé pour la définition de la syntaxe concrète de notre DSML.

Un fois que le modèle du domaine et la syntaxe concrète sont définis, il faut

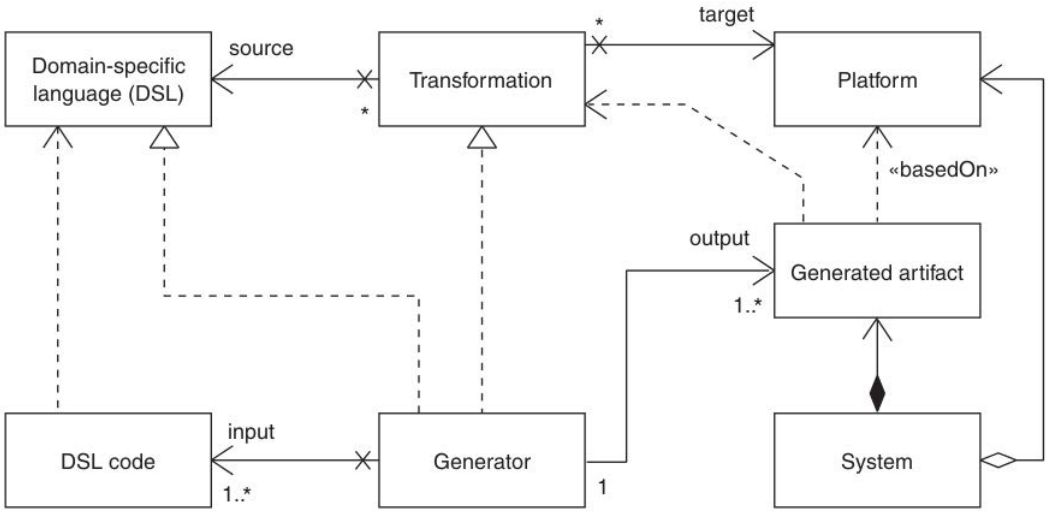

FIGURE III.7 – Intégration des plateformes - basé sur [2]

définir des règles de transformations qui permettront de faire le lien entre le modèle du domaine et les concepts des plateformes cibles.

3.3 Intégration des plateformes : Transformations et Génération de code

Comme nous l'avons indiqué précédemment, l'intégration des plateformes vient compléter la phase de conception en faisant le lien entre les concepts du domaine et les concepts des plateformes. Cette phase est aussi appelée phase d'implémentation ; elle consiste à mettre en place un compilateur qui traduit les programmes du DSL en une séquence d'appels de librairies [14].
Des transformations sont définies pour transformer les modèles définis à l'aide de la syntaxe concrète du DSML vers d'autres modèles ou vers un langage de programmation (correspondant à une plateforme cible) comme le montre la figure III.7.

Il existe alors deux types de transformations :

- De modèle vers modèle : elles consistent à convertir un modèle en un autre modèle du même système.
- De modèle vers texte : elles consistent à convertir un modèle en code qui lui correspond.

3.4 Utilisation

L'étape d'utilisation consiste à représenter les programmes souhaités par l'utilisateur final et à les compiler [14].
Dans notre cas, cette étape consiste à modéliser le système souhaité et à générer le code correspondant. Le code doit être compilable et exécutable sur la ou les plateformes cibles. L'utilisation d'un DSL ne se restreint pas seulement à la modélisation et la génération de code. On peut faire également des analyses sur les modèles pour valider les fonctionnalités du système dès les premières phases de conception.

4 Les DSML pour la robotique

Dans cette section, nous présentons tout d'abord les exigences que doivent respecter les DSML pour la robotique, nous présenterons ensuite les travaux qui ont adopté les techniques d'ingénierie dirigée par les modèles comme solution aux problèmes constatés dans le développement des applications de robotique et nous conclurons par une synthèse de ces travaux.

4.1 Exigences des langages de domaine pour la robotique

En plus des caractéristiques des DSL présentées précédemment, idéalement un DSL pour la robotique respecterait les caractéristiques suivantes [74].

1. **Facilité d'utilisation.** L'utilisation du DSL devrait être non seulement à la portée des experts en programmation mais aussi à la portée des experts en robotique et idéalement à la portée de simples utilisateurs de logiciels de robotique.
2. **Spécification d'une architecture à base de composants.** En supposant que la majorité des plateformes de robotique sont actuellement basées sur une architecture à base de composants, le DSL devrait permettre la spécification d'architectures à base de composants de systèmes de robotique autonomes.
3. **Spécification des comportements des composants**. Le DSL doit permettre la spécification des différents types de contrôle des composants. Il devrait être possible d'exprimer le contrôle sous forme algorithmique ou sous forme de machine à états finis.

4. **Neutralité vis à vis des architectures de robotique.** Le DSL ne devrait pas imposer une architecture de robotique particulière : délibérative, hybride, réactive ou basée sur le comportement (voir section 3.3).
5. **Plusieurs plateformes cibles hétérogènes.** Les composants de l'application devraient être indépendants des plateformes cibles et devraient pouvoir s'exécuter sur des robots réels ou des simulateurs. De plus, il devrait être possible de déployer certains composants générés à partir du DSL sur une plateforme et les autres composants sur une autre plateforme.
6. **Indépendance vis à vis des plateformes cibles.** Même si l'indépendance par rapport aux plateformes cibles est difficile à réaliser, le DSL devrait être aussi indépendant que possible des spécificités des platesformes d'exécution.
7. **Prise en charge de l'évolution des plateformes** La génération de code doit être aussi agile que possible de façon à ce que si les plateformes cibles évoluent, l'impact de leur évolution n'affecte pas énormèment le code des générateurs. De plus, idéalement, si des nouvelles plateformes sont ajoutées, elles pourraient réutiliser les transformations déjà définies.
8. **Evolution du DSL.** Idéalement, il devrait être possible de changer au moins certains aspects du DSL sans avoir à construire une nouvelle implémentation. L'évolution du DSL peut faire partie du processus du développement. En effet, si le processus de développement du DSL est itératif, les concepts du domaine sont adaptés ou renommés selon les besoins des plateformes et des applications. Une autre façon de gérer l'évolution des DSL consisterait de partir d'exemples concrets pour la définition des concepts du DSL, la génération de code, etc. De cette façon, le DSL conçu permettra le développement de ces applications.
9. **Abstractions du matériel** Idéalement, le DSL offrirait un ensemble d'abstractions du matériel sous forme de librairies afin de rendre les modèles définis indépendants du matériel du robot en plus d'être indépendant des plateformes.

4.2 Travaux existants

De nos jours, ils n'existe pas énormément de travaux qui mettent en oeuvre les techniques de l'ingénierie dirigée par les modèles pour la conception des DSLs pour la robotique. Certains travaux, notamment [75] et [76] ont appliqué ces techniques dans le cadre de l'étude d'un robot particulier ou d'une architecture de robotique

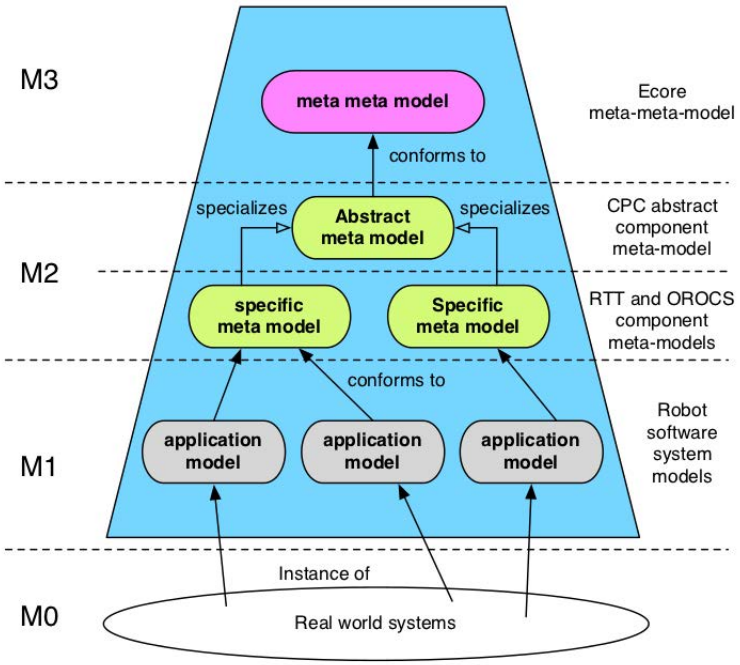

Figure III.8 – Les concepts de l'OMG appliqués à l'approche BCM - (tiré de [3])

particulière. Dans cette section, nous nous intéressons aux travaux qui ont défini des DSML en traitant les aspects de la variabilité en robotique (à travers des abstractions) et leur indépendance des plateformes cibles.

4.2.1 BRICS Component Model (BCM)

4.2.1.1 Description BRICS [77] est un projet européen qui vise à structurer et à formaliser le processus de développement des applications robotiques en fournissant des outils, des modèles et des librairies permettant d'accélérer le processus de développement. BRICS s'appuie sur les techniques d'ingénierie dirigée par les modèles afin de fournir un modèle appelé BCM (BRICS Component Model) [3] indépendant des plateformes à partir duquel une génération automatique de code est réalisée vers des plateformes de robotique (voir figure III.8). Le niveau M3 représente le méta-méta-modèle Ecore. Il s'ensuit un méta-modèle minimal et général de composants, indépendant des plateformes, de niveau M2 appelé CPC (Component Port Connector). Au même niveau, des méta-modèles spécifiques aux plateformes ROS et OROCOS sont définis. Au niveau M1, on retrouve des modèles instances

des méta-modèles du niveau M2.
Le méta-modèle CPC est spécialisé par les modèles des plateformes comme ROS et OROCOS. BRIDE [78] est le générateur de code qui assure les transformations de modèle vers modèle (du modèle instance du méta-modèle général vers le modèle instance du méta-modèle spécifique à la plateforme cible) ainsi que des transformations modèle vers texte (du modèle instance du méta-modèle spécifique à la plateforme cible vers la plateforme cible).

Le méta-modèle CPC est présenté dans la figure III.9, il spécifie qu'un `System` peut avoir plusieurs connecteur(s) (méta-classe `Connector`) et des composants (méta-classe `Component`). Un composant peut avoir des ports (méta-classe `Port`), une ou plusieurs propriétés (`Property`) et des sous-composants. Les abstractions représentées dans ce méta-modèle reflètent l'architecture des plateformes cibles (c.-à-d. une architecture à base de composants) mais ne sont pas représentatives des concepts du domaine. En dehors du BCM, le projet BRICS propose une séparation des préoccupations entre les composants de différentes natures à travers les *5Cs* :

- Computation : C'est la fonctionnalité responsable de l'accès aux données en lecture ou en écriture entre les différents composants, de leurs synchronisation et de leurs traitements.
- Coordination : Cette fonctionnalité détermine comment les composants d'un système doivent travailler ensemble. Elle détermine le rôle d'un composant dans un système.
- Composition : C'est la fonctionnalité qui permet de définir des composants composites.
- Communication : Se charge de l'envoi de données aux composants de computation.
- Configuration : Cette fonctionnalité permet de configurer les composants de communication et de computation.

Cependant, cette architecture n'a pas été intégrée au BCM car les auteurs argumentent que cette distinction n'est pas effectuée au niveau des plateformes cibles.

4.2.1.2 Discussion Le BCM permet une représentation de modèles à bases de composants et une génération automatique de squelettes de code vers ROS et OROCOS. Cependant, les abstractions du domaine ne sont pas représentées et la distinction entre les différentes natures des composants (c.-à-d. contrôle et matériel)

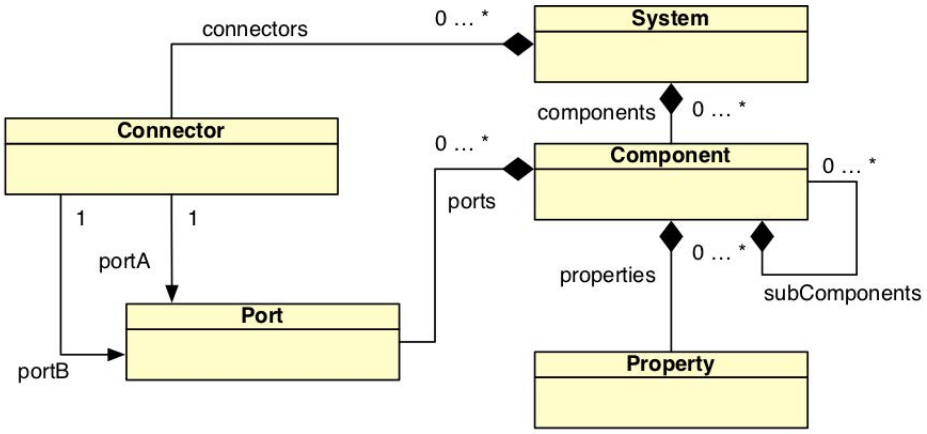

Figure III.9 – Le méta-modèle CPC - (tiré de [3])

d'une application de robotique n'a pas été intégrée au BCM. Les aspects comportementaux des composants notamment, les machines à états finis et les algorithmes ne sont pas représentés dans le BCM. De plus, la génération de code est effectuée vers une et une seule plateforme cible à la fois ; la génération de code vers plusieurs plateformes cibles hétérogènes n'est pas supportée.
L'évolution du BCM n'est pas envisageable éventuellement pour une intégration des 5Cs car il n'y aurait pas de correspondance avec les plateformes cibles [3].

4.2.2 Open Robot Controller Computer Aided Design (ORCCAD)

4.2.2.1 Description ORCCAD est un framework pour la spécification de la partie contrôle et commande d'un système de robotique [79, 80] destiné aux applications de robotique temps-réel. Le but d'ORCCAD consiste à fournir un ensemble d'outils qui permettent d'aider l'utilisateur tout au long du processus de conception, de vérification et de développement de son application. Cet ensemble d'outils, appelé CASE (Computer Aided Software Engineering), a été défini en utilisant les outils du projet de modélisation d'Eclipse [60] qui se basent sur les techniques d'ingénierie dirigée par les modèles.

L'approche ORCCAD utilise deux niveaux d'abstractions : un niveau fonctionnel et un niveau de contrôle [4]. Le niveau fonctionnel permet de représenter les tâches élémentaires du robot (*Robot Task (RT)*). Celles-ci sont décrites sous forme de comportements spécifiés dans des fichiers source écrits en ESTEREL [81] (voir figure III.10). Le niveau de contrôle (*Robot Procedure (RP)*) décrit ensuite la composition

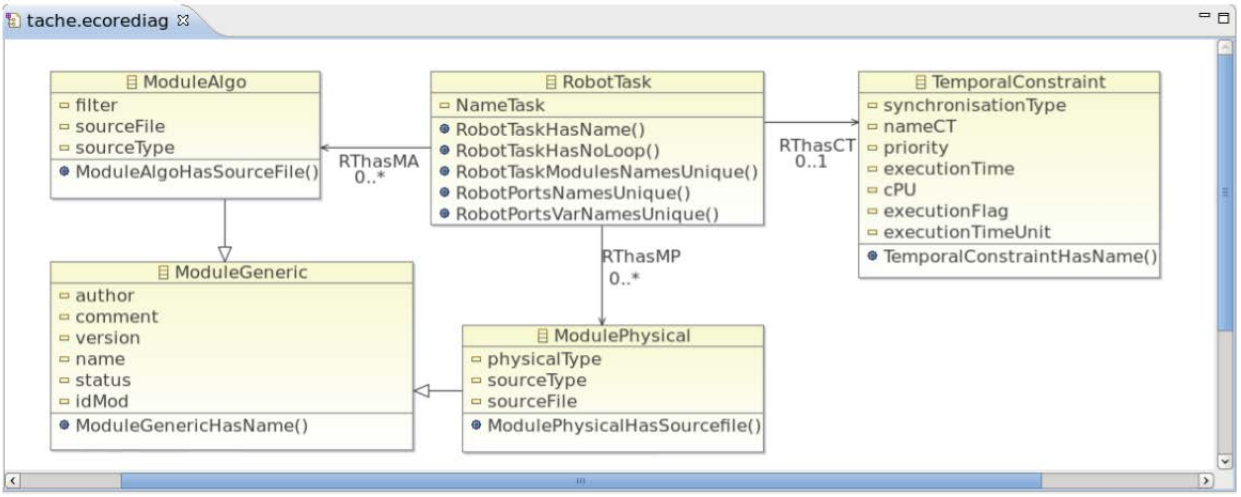

Figure III.10 – Le méta-modèle RT extrait du méta-modèle d'ORCCAD - (tiré de [4])

hiérarchique des RTs. Les abstractions utilisées sont suffisament de haut niveau pour ne pas être spécifiques à une plateforme particulière et pour représenter les aspects temps-réel requis pour une application de robotique. A partir des modèles réalisés avec l'éditeur graphique d'ORCCAD (instance du méta-modèle dont une partie a été présentée précédemment), le générateur de code fournit du code C++, à partir des RTs et RPs, ainsi que le *glue code* qui permet de les connecter avec le système d'exploitation utilisé.

4.2.2.2 Discussion Malgré les avantages d'utilisation d'ORCCAD notamment la généricité de ses modèles et leur indépendance des détails des plateformes cibles, ORCCAD ne définit pas des abstractions du matériel et ne prend pas en charge la génération de code vers d'autres plateformes cibles de robotique. L'une des solutions possibles pour assurer la prise en charge de l'hétérogénéïté des plateformes consisterait à mettre en oeuvre un générateur de code à partir du code C++ généré vers d'autres plateformes. De plus, ces middlewares doivent prendre en charge l'aspect temps-réel, il n'est donc pas possible de générer du code vers des middlewares tel que ROS qui ne prend pas en compte cet aspect.

4.2.3 SmartSoft

4.2.3.1 Description SmartSoft [82][83] est un framework qui offre un ensemble de patterns de communication générique qui permettent l'assemblage et la composi-

tion des composants grâce à des interfaces structurées et consistantes correspondant à une sémantique bien définie. Ces interfaces sont basées sur des services de communication standard (client/server, master/slave, publish/subscribe, request/response, etc.). Le modèle de composants comporte également un automate générique qui gère le cycle de vie d'un composant [84].

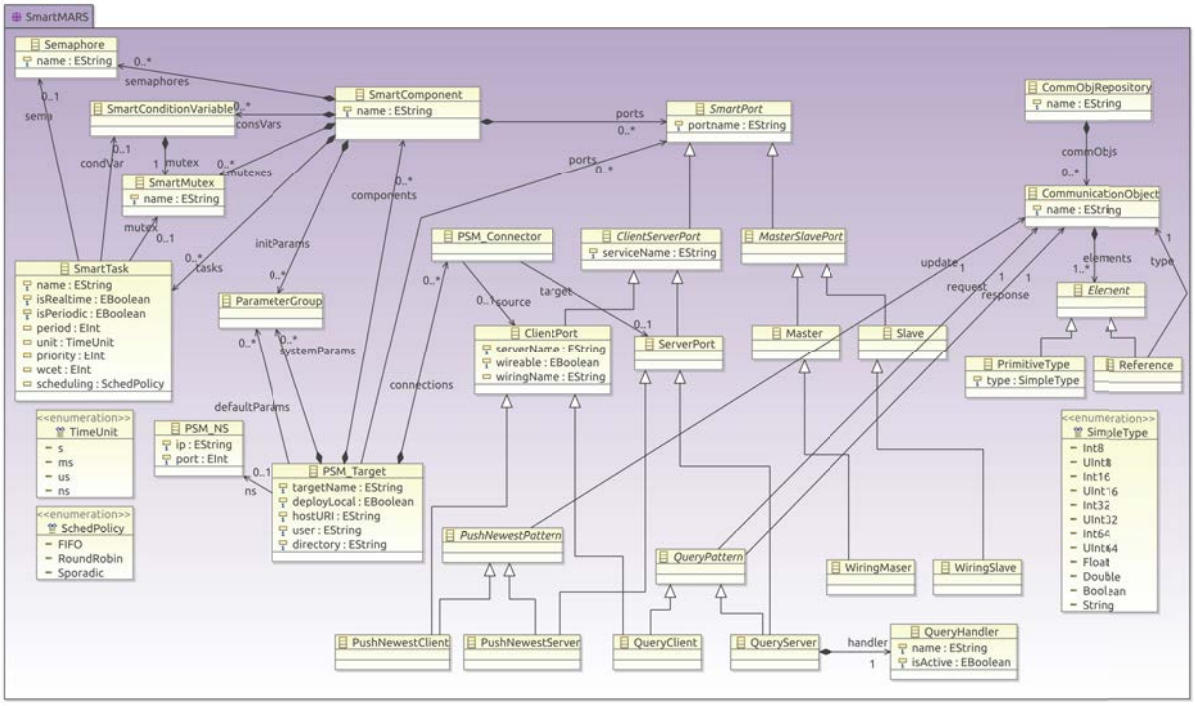

Figure III.11 – Le méta-modèle de SmartSoft - (tiré de [5])

Le modèle de composants SmartSoft et ses mécanismes de communication ont été définis dans un méta-modèle appelé SmartMars (Modeling and Analysis of Robotic Systems) [16]. Les patterns de communication produisent différents modes de communication comme la communication dans une seule direction ou l'intéraction request/response. Par exemple, le pattern send est un mode de communication Client/Serveur dans une seule direction. Le pattern Query est un mode de communication Client/Serveur qui utilise l'intéraction request/response. Ces concepts sont représentés dans le méta-modèle SmartMars à travers les méta-classes `QueryPattern`, `PushNewestPattern`, `ClientServerPort`, `MasterSlavePort`, et `CommunicationObject`. Deux types de méta-modèles sont définis dans le méta-modèle SmartMars (voir figure III.11) : un méta-modèle indépendant des plateformes représentant les patterns de communication de SmartSoft ainsi que la structure générale des composants et

leur cycle de vie et un méta-modèle spécifique aux plateformes permettant de spécifier les informations relatives à la plateforme choisie (méta-classes `PSM-Connector`, `PSM-Target` et `PSM-NS` dans la figure III.11).
Le méta-modèle SmartMars a été implémenté sous forme de profil UML constituant ainsi une base de développement pour un outil intégré à Eclipse appelé SmartMDSD (Model Driven Software Design). SmartMDSD offre une chaîne d'outillage de la modélisation jusqu'à la génération de code (vers CORBA et ACE).

4.2.3.2 Discussion SmartSoft permet la modélisation à base de composants des applications de robotique indépendamment des plateformes cibles et permet également une spécification claire et concise des concepts de communication entre les différents composants définis. Les patterns de communication exprimés sont stables et ont été utilisés en robotique depuis plus de dix ans. Le méta-modèle de SmartSoft permet aussi de représenter le cycle de vie d'un composant à l'aide d'un automate où les états initialisent le composant, l'exécutent et peuvent indiquer l'échec de son exécution.
Cependant, il n'y a pas de disctinction entre la nature des composants utilisés. En d'autres termes, on ne sait pas si ces composants sont matériel ou logiciels. Par conséquent, la prise en charge de plusieurs plateformes cibles hétérogènes n'est pas possible (c.-à-d. si on souhaite générer du code vers un simulateur et un middleware). De plus, les abstractions du domaine ne sont pas suffisantes car il n'est pas possible d'exprimer les abstractions du matériel par exemple. Concernant l'évolution des plateformes cibles et du méta-modèle, ces points n'ont pas été abordés dans les travaux de SmartSoft.

4.2.4 The 3 View Component Meta-Model (V^{3}CMM)

4.2.4.1 Description V^{3}CMM [6] s'appuie sur les concepts de l'OMG afin de fournir un méta-modèle indépendant des plateformes pour la conception des applications de robotique à base de composants. Ses principales caractéristiques sont la simplicité et l'économie des concepts d'un côté et la réutilisabilité des composants de l'autre. V^{3}CMM utilise certains concepts d'UML afin d'offrir trois vues principales : (1) une vue structurelle pour la description de la structure des composants et de la communication entre eux (2) une vue de coordination décrivant le comportement de chaque composant sous forme de machine à états finis (en réutilisant les concepts

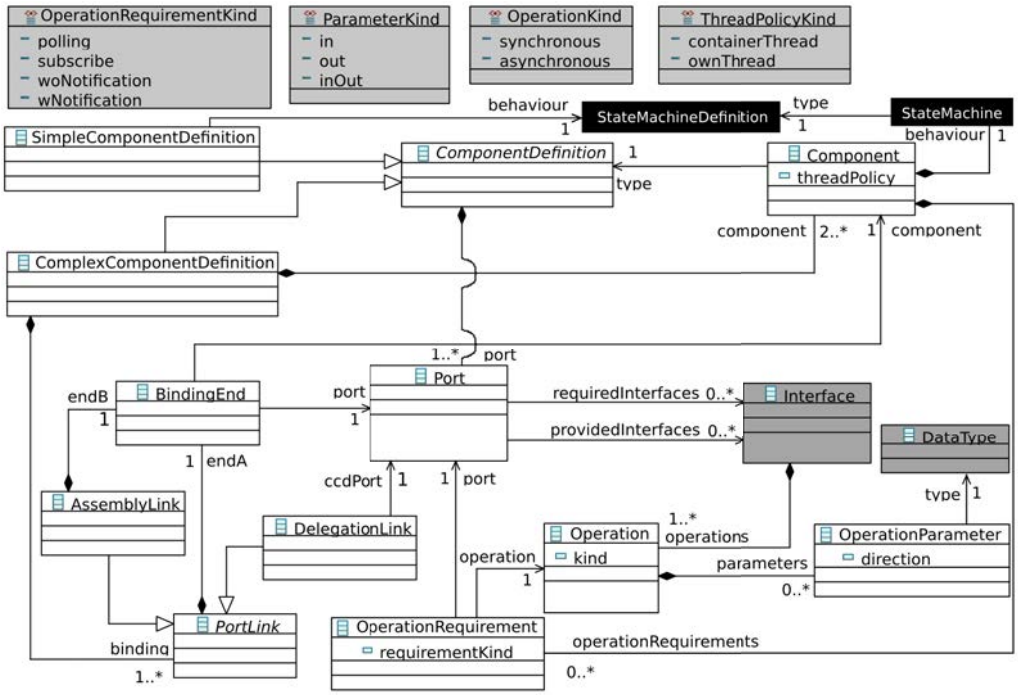

Figure III.12 – Méta-modèle V3CMM - tiré de [6]

d'UML) (3) une vue algorithmique pour la description de l'algorithme de chaque composant selon l'état dans lequel il se trouve (qui reprend une représentation simplifiée du diagramme d'activités d'UML). Un extrait de ce méta-modèle est présenté dans la figure III.12. Une chaîne d'outillage a été développée avec les outils de modélisation d'Eclipse permettant ainsi la génération de code en deux étapes. Tout d'abord des transformations de modèle-vers-modèle sont définies entre un modèle instance du méta-modèle V^{3}CMM et un modèle instance du méta-modèle représentant les concepts de la programmation orientée objets. Puis, à partir du deuxième modèle, des transformations de modèle vers texte sont définies vers le langage ADA.

4.2.4.2 Discussion Il est à noter que même si V^{3}CMM ne contient pas les concepts de la robotique, il a été testé et essentiellement utilisé en robotique et dans le domaine des réseaux de capteurs sans fils [85] et d'autres domaines comme la domotique. Actuellement, V^{3}CMM a été étendu afin de permettre la spécification des contraintes temps-réel. Des éditeurs textuels avec une validation de modèle V^{3}CMM ainsi que plusieurs implémentations vers différents frameworks cibles.
Cependant, V^{3}CMM ne fait pas distinction entre les composants matériel et logiciel d'une application. Ainsi, il n'est pas possible de générer du code vers plusieurs pla-

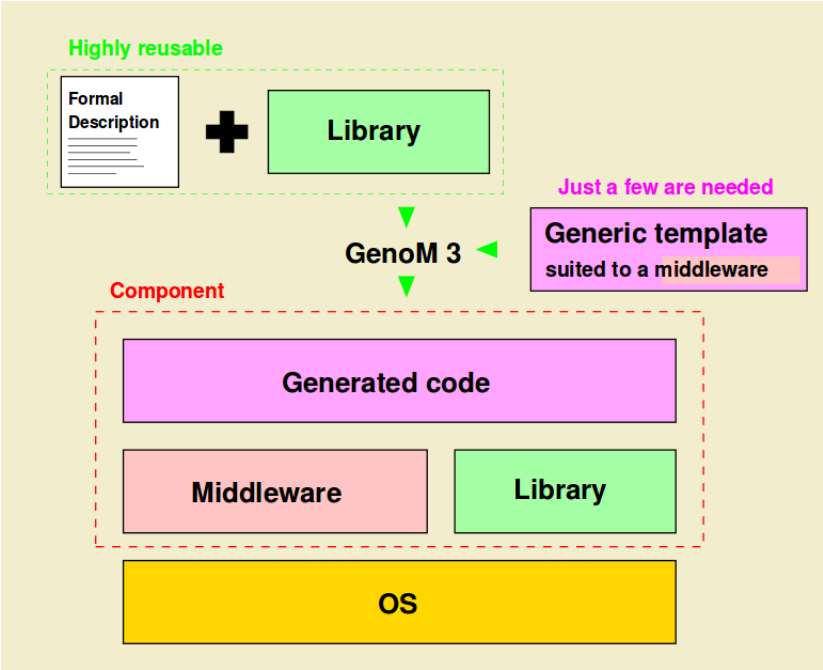

FIGURE III.13 – Architecture de Genom3

teformes cibles hétérogènes. De plus, il n'est pas possible d'exprimer les abstractions du matériel de haut niveau. L'évolution du méta-modèle et des plateformes cibles n'a pas été abordée dans les travaux V^{3}CMM.

4.2.5 GenoM3

4.2.5.1 Description GenoM3 [86] a été défini dans le but d'assurer l'indépendance des composants logiciels d'une application des middleware de robotique. L'idée principale consiste à découpler la structure des composants de leurs noyaux algorithmiques. Cette idée a été réalisée en mettant en oeuvre un template par middleware pour la définition des squelettes des composants dans celui-ci. Ce template est organisé sous forme d'un fichier source. En plus de ce template, un DSL textuel a été défini permettant la description des composants et de leurs fonctionnalités, notamment les ports, les types de données, les services, les opérations (définies sous forme de *codels* [87]) et le contexte d'éxécution (task ou thread), sous forme de spécification formelle (voir figure III.13). Un interpréteur se charge ensuite d'instancier les composants à partir de leurs descriptions conformément au template du middleware choisi. Du code exécutable est alors généré correspondant aux composants décrits.

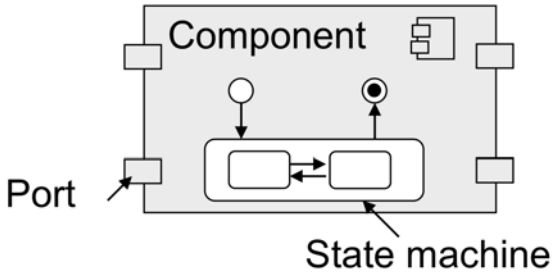

Figure III.14 – Composant RTC

4.2.5.2 Discussion À travers plusieurs études de cas, il a été prouvé que l'indépendance des plateformes est garantie à travers l'utilisation de GenoM3 [88].
Cependant, l'architecture d'une application doit être conforme à la structure des templates définis et la spécification formelle des composants n'est pas au même niveau d'abstraction que les méta-modèles. La transformation sous forme de template de composants ne correspond pas aux meilleures pratiques du développement dirigé par les modèles [16] (MDSD : Model Driven Software Development). De plus, la génération de code vers des plateformes hétérogènes n'est pas prise en charge. De plus l'aspect abstraction des données du matériel n'est pas pris en charge.

4.2.6 Robot Technology Component (RTC)

4.2.6.1 Description OpenRTC [89] a été conçu par l'OMG afin de proposer un standard pour la robotique. Plusieurs plateformes se basent sur le standard RTC et utilisent son modèle de composants comme OpROS [90–92] et RTM [93].
Un RTC est une représentation logique du matériel et/ou d'une entité logicielle qui fournit des fonctionnalités et des services souvent utilisés.
RTC étend les fonctionnalités des composants proposés par UML et s'appuie sur les technologies de la MDA pour proposer un modèle indépendant des plateformes exprimé en UML et composé de trois parties :

- *Lightweight RTC* : un modèle basique contenant la définition de concepts simples comme composant et port comme le montre la figure III.14. Il définit des interfaces avec des abstractions et des stéréotypes.
- *Execution semantics* : C'est une extension du modèle basique (Lightweight RTC) qui supporte les patrons de conception critiques de communication utilisés en robotique.

– *Introspection* : Une API qui permet d'examiner les composants, leurs ports et leurs connexions à l'exécution.

Trois modèles spécifiques aux plateformes (PSM) ont ensuite été définis dans le langage IDL (Interface Definition Language) de l'OMG et ont été proposés afin de spécifier les mécanismes de communication :

– *Local* : indique que les composants communiquent entre eux et qu'ils sont placés dans le même réseau.
– *Lightweight CCM (Corba Component Model)* Les composants sont distribués et communiquent via l'interface CCM-based middleware.
– *CORBA* Les composants sont distribués et communiquent via CORBA.

En plus des composants *LightWeight RTC*, RTC définit trois autres types de composants :

– Le traitement des flots de données (*Data flow processing/Periodic sampled data processing*) : correpond à l'exécution de type périodique. Les composants de type flots de données sont exécutés périodiquement. Un composant de ce type peut aussi être un composant composite contenant d'autre composants de flots de données. De cette façon, le traitement de données peut être décomposé de manière hiérarchique.
– Le traitement Stimulus/Réponse appelé également traitement de données asynchrone. Les applications utilisant ce type de composant doivent généralement réagir aux changements qui surgissent dans l'environnement du robot. Le comportement est représenté sous forme de machine à états finis (FSM). Lorsqu'un événement est signalé, la FSM change d'état en exécutant l'action associée à la transition par laquelle elle est passée.
– Modes d'opération : fournit un support pour les applications qui doivent naviguer entre différentes implémentations de la même fonctionnalité.

4.2.6.2 Discussion Le modèle de composant RTC est considéré comme une spécification avancée proposée par l'OMG dans le domaine de la robotique. Ce modèle a servi de standard pour beaucoup de plateformes de robotique : RTM (Robot Technology Middleware) [93], OPRoS [90], GostaiRTC [94], etc.

Cependant, la spécification de RTC est fortement influencée par des architectures qui se basent sur un échange de flots de données. Par conséquent, le modèle de composant RTC est influencé par un automate interne fortement lié à un modèle d'activités à l'intérieur d'un composant. Il ne permet pas la spécification de plusieurs

tâches à l'intérieur d'un composant. De plus, les composants de natures différentes ne sont pas distingués, il n'est pas donc pas possible de générer du code vers plusieurs plateformes hétérogènes. L'évolution du DSL et des plateformes cibles n'a pas été abordée pour le modèle RTC. Par ailleurs, les abstractions du matériel ne sont pas prises en charge.

4.3 Synthèse

Les DSML existants pour la robotique répondent partiellement aux exigences présentées précédemment comme le montre le tableau III.4. BCM [3], SmartSoft [16], V^{3}CMM [6], Ge^{n}oM3 [86] et RTC [89] permettent la représentation d'une architecture à base de composants contrairement à ORCCAD [4] qui spécifie plutôt le niveau fonctionnel (Robot Task) et le niveau contrôle (Robot Procedure) d'une application. L'indépendance par rapport aux plateformes cibles a été respectée dans les DSML existants soit à travers des modèles indépendant des plateformes comme le cas de BCM, SmartSoft, V^{3}CMM, ORCCAD et RTC ou à travers des templates comme le cas de Ge^{n}oM3. Concernant la neutralité vis-à-vis des architectures de robotique elle est également gérée dans les travaux existants vu qu'aucun concept spécifique à une architecture particulière n'est spécifié.
En revanche, la génération de code vers plusieurs plateformes cibles hétérogènes (c.-à-d. simulateur et middleware ou plusieurs middleware), la prise en charge de l'évolution des plateformes cibles, l'évolution du DSL ainsi que la représentation des abstractions du matériel ne sont pas actuellement prises en charge dans les travaux existants.

5 Conclusion

Dans ce chapitre, nous avons présenté les DSML et leur cycle de vie. Nous avons ensuite étudié les travaux existants mettant en oeuvre les techniques d'ingénierie dirigée par les modèles afin de répondre aux problèmes constatés dans le développement des applications de robotique dûs à la variabilité du matériel, des algorithmes et des middleware et permettre ainsi une facilité d'utilisation, de développement et de réutilisabilité des applications.
Nous avons vu que ces travaux ne gèrent pas l'hétérogénéïté des plateformes cibles. La distinction entre les composants matériel et les composants logiciels n'est pas

effectuée au niveau de la modélisation. De plus, les abstractions du domaine définies sont insuffisantes car elles sont très générales, elles sont souvent basées sur l'architecture des plateformes cibles (architecture à base de composants). SmartSort se distingue par la stabilité des patterns de communication qu'il propose et donc par des abstractions pour la communication basées sur une expertise du domaine.
Certes ces DSML traitent le problème de l'indépendance par rapport aux plateformes cibles mais il devrait être possible de manipuler les concepts de robotique tels qu'ils sont utilisés par les roboticiens en fixant tout d'abord une terminologie du domaine définie par des roboticiens. De plus, il devrait être possible de générer du code vers plusieurs plateformes supportant ou non les aspects temps-réels.
Dans le chapitre suivant, nous présentons notre DSML pour la robotique : RobotML et les solutions que nous proposons afin de répondre aux exigences présentées dans ce chapitre.

Caractéristiques	V^3CMM [6]	SmartSoft [16]	ORCCAD [4]	BCM [3]	Ge^noM3 [86]	RTC [89]
Facilité d'utilisation	Oui	Oui	Oui	Oui	Oui	Oui
Spécification d'une architecture à base de composants	Oui	Oui	Non	Oui	Oui	Oui
Spécification du comportement des composants	Oui	Non	Oui	Non	Oui	Oui
Indépendance des plateformes cibles	Oui	Oui	Oui	Oui	Oui	Oui
Neutralité vis-à-vis des architectures de robotique	Oui	Oui	Oui	Oui	Oui	Oui
Plusieurs plateformes cibles hétérogènes	Non	Non	Non	Non	Non	Non
Prise en charge de l'évolution des plateformes	Non	Non	Non	Non	Non	Non
Evolution du DSL	Non	Non	Non	Non	Non	Non
Abstractions du matériel	Non	Non	Non	Non	Non	Non

Table III.4 – Comparaison entre les DSML pour la robotique existants

Deuxième partie

Contributions

Robotic Modeling Language (RobotML)

1 Introduction

Notre étude des travaux existants a montré que les concepts utilisés dans les DSML de robotique actuels sont insuffisants pour représenter tous les aspects d'une application de robotique. D'une part les DSML existants se concentrent sur la représentation de la partie architecturale ou (et) celle de contrôle et de communication d'une application en utilisant des modèles à base de composants généralistes qui ne se basent pas sur des abstractions du domaine. Par conséquent, ces abstractions ne permettent pas la gestion de la variabilité du matériel ou l'expression de données abstraites. De plus les travaux existants ne prennent pas en charge plusieurs plateformes cibles hétérogènes (de natures différentes : simulateurs et middlewares).

Afin de faciliter le développement des applications de robotique, il devrait être possible pour un roboticien de pouvoir manipuler des concepts qu'il a l'habitude d'utiliser à travers des modèles et des outils qui leurs sont associés. Ces concepts peuvent être extraits à partir de systèmes existants telles que les ontologies qui représentent des bases de connaissance du domaine.
Il devrait aussi être possible d'exprimer le comportement d'un composant à un niveau plus abstrait que le code à travers des automates par exemple. Enfin, les concepts d'un DSML devraient être suffisamment génériques pour prendre en charger plusieurs plateformes cibles hétérogènes.
Nous avons ainsi contribué au développement d'un DSML pour la robotique : RobotML qui offre un langage axé sur des abstractions du domaine permettant tout d'abord de réduire le fossé sémantique entre les roboticiens et les développeurs. Ces abstractions sont basées sur une ontologie du domaine définie par des experts de

robotique.
De plus, RobotML offre une chaîne d'outillage allant de la modélisation de scénarios de robotique jusqu'à la génération de code vers plusieurs plateformes cibles hétérogènes facilitant ainsi le développement des applications de robotique.
Les composants de différentes natures sont distingués dans RobotML. Par exemple, on distingue les composants matériels des composants de contrôle et des composants de déploiement.
RobotML permet :

- la spécification de l'architecture d'un système de robotique : composants de contrôle, composants matériel, leurs ports et les types de données qu'ils s'échangent.
- la spécification de la communication entre les composants à travers des ports de flots de données ou des ports de service. Il est également possible de préciser le type de communication entre les composants (synchrone ou asynchrone).
- la spécification du comportement des composants de l'architecture du système à travers des machines à états ou des algorithmes.
- la spécification d'un plan de déploiement qui permet de définir plusieurs plateformes cibles hétérogènes (simulateurs et middleware).

RobotML a été développé dans le cadre du projet ANR PROTEUS (Plateforme pour la Robotique Organisant les Transferts Entre Utilisateurs et Scientifiques) [95]. PROTEUS compte quinze partenaires notamment, DASSAULT Aviation [96], ECA [97], CEA [98], GOSTAI [99], Intempora [100], THALES [101], LASMEA [102], TOSA [103], GREYC [104], INRIA [105], ONERA [106], PRISME [107], EFFIDENCE [108], WIFIBOT [109] et enfin le LIP6 [110] au sein duquel cette thèse a été réalisée.
Notre contribution dans le cadre du projet PROTEUS a consisté à participation aux différentes phases de développement de RobotML et à la définition des parties contrôle et communication de RobotML. De plus, nous avons défini un générateur de code à partir de RobotML vers le middleware OROCOS.
Dans ce chapitre, nous décrivons le cycle de vie de RobotML et sa chaîne d'outillage. Nous présenterons enfin un exemple de scénario défini avec RobotML.

2 Vue d'ensemble sur le cycle de vie de RobotML

Comme nous l'avons présenté dans le chapitre 3, le cycle de vie d'un DSML comporte quatre phases : l'analyse du domaine, la conception, l'intégration des plateformes cibles à travers la définition de générateurs de code et enfin l'utilisation. Pour effectuer l'*Analyse du domaine*, nous nous sommes basés sur un état de l'art sur les middleware, les simulateurs et les DSL existants pour la robotique. Cet état de l'art a permis d'identifier certaines exigences notamment l'indépendance par rapport aux plateformes cibles (middleware et simulateurs) et la représentation d'une architecture à base de composants. Ces dernières, combinées avec les concepts du domaine fournis par une ontologie de la robotique mobile définie dans le cadre du projet PROTEUS, ont permis de délimiter le domaine d'application (c.-à-d. la robotique mobile) et d'extraire de l'ontologie les abstractions dont les roboticiens ont besoin pour la définition de leurs applications.
Après cette étape d'analyse du domaine arrive l'étape de *Conception* où l'on identifie la syntaxe abstraite ainsi qu'une syntaxe concrète qui s'articule autour de cette syntaxe abstraite afin de fournir un éditeur graphique permettant de manipuler les concepts du domaine.
Plusieurs générateurs de code ont ensuite été implantés afin d'établir des règles de transformations à partir de ces abstractions vers les plateformes cibles.
Enfin, la phase d'utilisation de RobotML consiste à tester toute la chaîne d'outillage de la modélisation vers la génération de code.

3 Analyse du domaine

L'analyse du domaine pour la définition de notre DSML se base sur une ontologie de la robotique mobile définie dans le cadre du projet PROTEUS [7] ainsi que sur l'état de l'art des simulateurs et des middleware robotique. Une ontologie est une représentation formelle des connaissances décrivant un domaine donné [111].
L'ontologie de PROTEUS a été construite à partir de la connaissance des experts en robotique dans différents sous-domaines. Leur expertise porte sur les domaines suivants : commande, contrôle, perception, navigation, localisation, contrôle du trafic, optimisation, planification et simulation. Afin de partager leurs connaissances, des experts du domaine partenaires du projet PROTEUS, se sont réunis pour décrire des scénarios et pour exprimer leurs connaissances et leurs besoins dans la robotique

mobile autonome. À partir de ces réunions, les exigences de l'ontologie ont été définies.
Ces exigences concernent la modélisation de la mécanique et des composants électroniques ainsi que des architectures de contrôle, les systèmes, les simulateurs, etc. Ces exigences incluent également la modélisation d'un robot donné ainsi que le suivi de son évolution (comportement) pendant l'exécution de son scénario. Comme il faut modéliser le comportement du robot, il est nécessaire de modéliser son environnement et sa mission.
L'ontologie a été définie autour du concept `System`. Ce dernier représente une entité ayant des intéractions, il peut être considéré comme un composant au sens du génie logiciel. Un `System` est défini comme un bloc qui déclenche des intéractions et est impacté par les intéractions des autres systèmes. Le choix d'une représentation d'une telle architecture a pour but de permettre la définition des composants, de leurs comportements et des différents échanges de données entre eux. Ainsi, le code final généré à partir du DSML (dont les concepts sont extraits de l'ontologie) pourra correspondre aux concepts architecturaux utilisés par les plateformes cibles.
L'ontologie de PROTEUS a été décrite avec le langage OWL que nous avons présenté dans la section 3.1.1.1 et est organisée de manière modulaire où les modules sont définis comme une spécialisation du module principal (le `Kernel`). Certains modules permettent de spécifier les classes décrivant les `System`, leurs ports et leurs caractéristiques. D'autres modules servent à décrire l'environnement du robot, les outils de simulation utilisés pour tester les solutions ou encore les expérimentations (validation, vérification, tests).
La figure IV.15 montre la classification des différents systèmes (`System`) définis par l'ontologie. Comme nous l'avons dit un `System` est une entité ou une unité logique qui réalise une intéraction, il peut lui même comporter d'autres `System`s dans ce cas, il s'agit d'un `CompositeSystem`. Les `PhysicalObject`, `Software` et `AtomicSystem` sont des `System`s. Un `PhysicalObject` décrit toutes les entités physiques dans un scénario de robotique. Il peut être un `Hardware` (c.-à-d. horloge, effecteur, un bus de communication ou encore un capteur) sur lequel le `Software` s'exécute. Un `PhysicalObject` peut également être un `Agent` capable d'agir comme les robots (`Robot`) et les humains (`Human`).

La figure IV.16 montre la relation entre les différentes `System`s. Une `Interaction` est un concept abstrait utilisé pour décrire les intéractions entre les `System`s. Elle s'effectue à travers des `Port`s qui représentent des interfaces pour les entrées/sor-

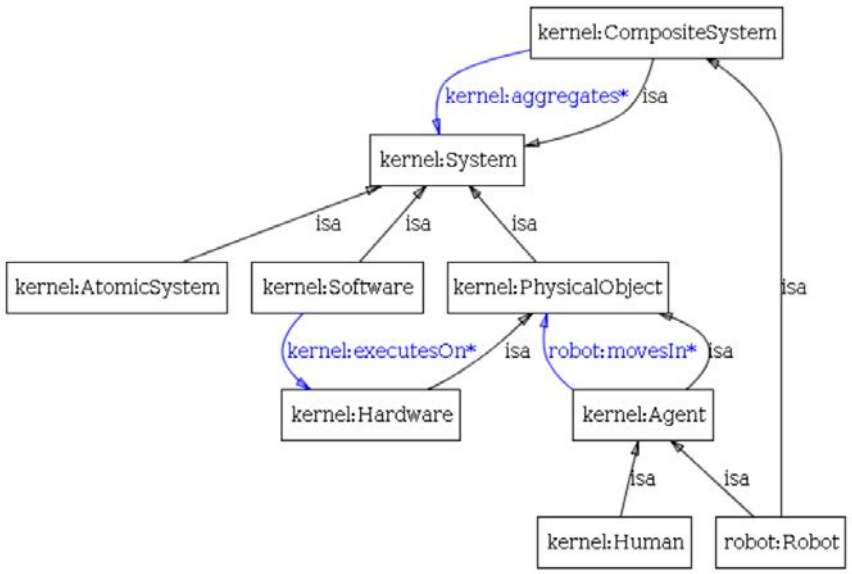

FIGURE IV.15 – Classification des systèmes dans l'ontologie (extrait de [7])

ties entre les `Systems`. Un `System` possède un état et un modèle d'évolution. Un `EvolutionModel` représente la propriété intrinsèque d'un composant d'évoluer à travers le temps. Deux types spécifiques d'`EvolutionModel` sont distingués mais ne sont pas représentés dans cette figure : `Algorithm` et `StateMachine`. Un `Algorithm` est une méthode pour la résolution d'un problème sous forme d'une séquence d'instructions. Une machine à états est définie par des transitions, des événements et s'appuie sur des états (`State`) pour indiquer l'évolution d'un système à travers des états.

Les données échangées entre les différents `Systems` sont représentées dans le module Information (voir figure IV.17). Une `Information` est directement ou indirectement liée à une intéraction. Le lien direct indique qu'une intéraction transmet des informations. Le lien indirect indique qu'une intéraction possède un protocole qui contient des informations. Une `Abstraction` est une `Information` qui n'est pas directement interprétée par la machine.
Contrairement aux abstractions, `Data` est une information directement interprétée par la machine. Une spécification de `Data` qui n'est pas représentée sur cette figure est `PhysicalData` ; elle désigne un objet mathématique-algébrique ayant une unité physique associée.

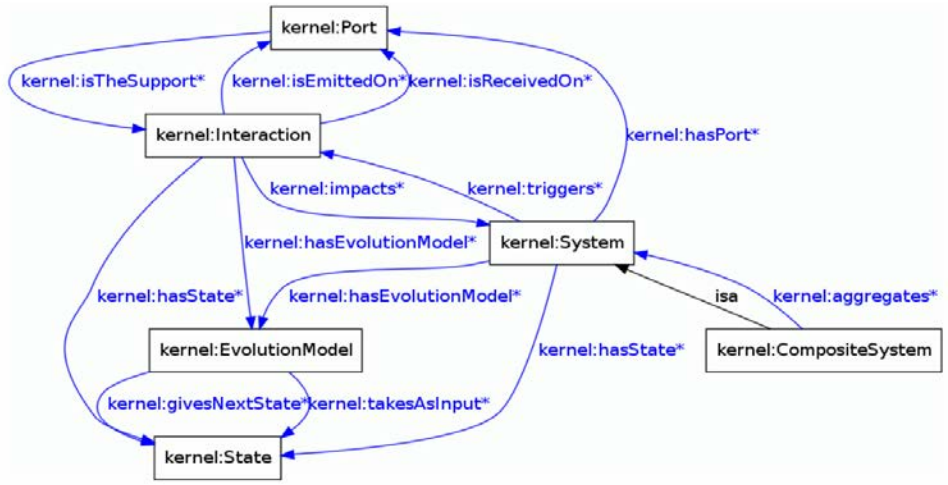

Figure IV.16 – Intéraction entre les systèmes dans l'ontologie (extrait de [7])

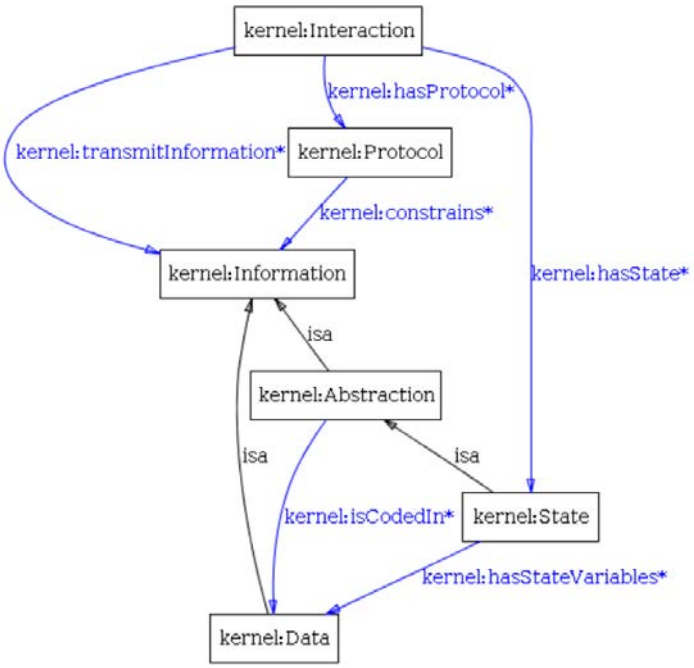

Figure IV.17 – La classe Information dans l'ontologie (extrait de [7])

Les classes de l'ontologie représentant les concepts du domaine peuvent être reprises, modifiées ou supprimées dans le DSML. Par exemple, `Interaction` permet de définir un lien entre deux `Systems` mais n'est pas assez explicite. Nous l'avons remplacé dans le modèle du domaine par les concepts `Port` et `Connector` pour cap-

turer les propriétés d'une intéraction.
Le principal avantage de cette démarche est de pouvoir réutiliser les bases de connaissance des experts du domaine pour enrichir le DSML [112]. L'ontologie constitue le point d'entrée du modèle du domaine de RobotML.
Dans le projet PROTEUS, le but de la définition de l'ontolgie dépasse la représentation des concepts du domaine. L'une des ambitions de son utilisation consiste à normaliser le domaine de la robotique mobile. Cet objectif n'est pas encore atteint car les abstractions définies dans l'ontologie sont basées sur les connaissances des roboticiens comme nous l'avons présenté précédemment ou sur des descriptions de scénarios qu'ils souhaitent représenter et non pas sur des exemples concrets. On ne sait donc pas comment implanter certaines de ces abstractions.
Dans la section suivante nous présentons la conception de RobotML qui se base sur les concepts de l'ontologie pour représenter les abstractions du domaine.

4 Conception de RobotML

La conception de RobotML est passée par les étapes suivantes :

1. Extraction des abstractions du domaine à partir de l'ontologie : Les exigences d'un langage de domaine pour la robotique présentées dans la section 4.1 du chapitre 3. sont extraites de l'ontologie d'une part et de l'état de l'art sur les simulateurs et les middleware d'une autre part.
2. Représentation de ces abstractions dans un modèle du domaine sous forme d'un méta-modèle puis dans un profil qui étend les concepts d'UML pour les adapter aux concepts du domaine.
3. Après la vérification et la validation de la syntaxe abstraite, définition de la syntaxe concrète sous forme d'un éditeur graphique basé sur le profil UML.

Dans cette section nous présentons toutes ces étapes.

4.1 Syntaxe abstraite

Le DSML doit correspondre aux concepts du domaine provenant de l'ontologie. Ceci constitue la première exigence du DSML afin d'utiliser une terminologie définie par les experts du domaine de la robotique. À partir de l'ontolgie, les concepts

spécifiques aux domaines sont alors extraits. Ces concepts sont filtrés pour n'en retenir que ceux dont on a besoin pour la définition d'une application de robotique : l'architecture (les composants, leurs ports, les types de données), la communication entre les composants ainsi que leurs comportements.
La syntaxe abstraite de RobotML a d'abord été représentée dans un modèle du domaine (méta-modèle) puis dans un profil qui étend les concepts d'UML pour les adapter aux concepts du domaine. Les concepts du modèle du domaine ont généralement le même nom que ceux de l'ontologie sauf si ces derniers ne sont pas assez explicites comme le cas du concept `Interaction` que nous avons remplacé par les concepts `Port` et `Connector`.
Étant donné que l'on souhaiterait que notre DSML supporte plusieurs plateformes cibles hétérogènes, les concepts du DSML sont également extraits à partir d'un état de l'art sur les simulateurs et les middlewares. Deux stratégies simples sont possibles pour extraire les concepts correspondant à plusieurs plateformes : faire l'union ou l'intersection des concepts de ces plateformes. Autrement il faudrait réaliser un travail considérable de factorisation et de construction de nouvelles abstractions ce qui n'a pas été possible dans Proteus malheureusement. Si nous choisissons l'intersection des concepts identifiés dans les plateformes, on imposerait des restrictions et les concepts de certaines plateformes ne pourraient pas être représentés. Si nous choisissons l'union, nous couvrons les concepts existants et aussi, il sera plus facile de supporter des nouvelles plateformes. Nous avons alors choisi l'union des concepts identifiés dans les plateformes. Cependant, le DSML devient ainsi plus complexe que les plateformes cibles et il est possible que des concepts ne soient pas traduisibles sur certaines plateformes
Le tableau IV.5 montre la correspondance entre les concepts utilisés pour définir l'ontologie en OWL (voir section 3.1.1.1 et les concepts d'UML que nous avons utilisés pour représenter notre modèle de domaine.

Classe OWL	Concepts d'UML
Concept	Class
subClassOf	Inheritence
Property	Association, Attribute
Property :IsA	Inheritence
Property :HasA	Composition
Cardinality	Multiplicity

Table IV.5 – Correspondance entre OWL et UML

Dans ce qui suit nous présentons notre modèle de domaine ainsi que le profil UML.

4.1.1 Le modèle de domaine RobotML : méta-modèle

L'objectif principal de notre DSML consiste à permettre aux roboticiens de manipuler les concepts dont ils ont l'habitude d'utiliser pour la définition de leurs applications. Nous avons distingué trois parties dans notre modèle de domaine : la partie architecture, la partie communication et la partie contrôle. Cette distinction découle des principales parties à définir pour une application de robotique.
La figure IV.18 montre une vue globale des principaux packages du modèle du domaine RobotML :

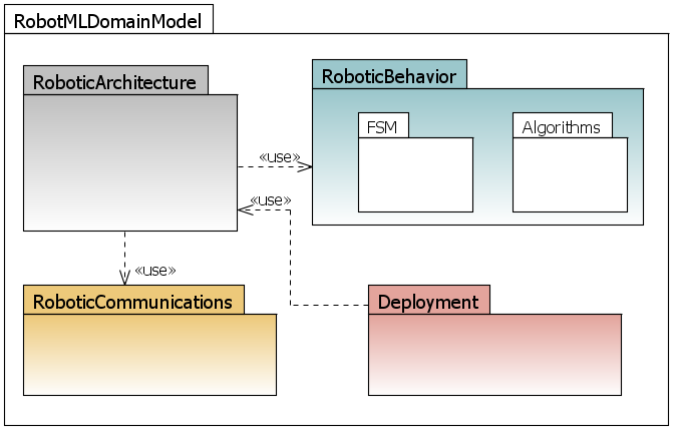

Figure IV.18 – RobotML Domain Model

- Le package `RoboticArchitecture` regroupe les concepts architecturaux et structurels qui définissent une application de robotique. Ce package représente la **partie Architecture**.
- Le package `RoboticBehavior` permet de définir le comportement d'un composant d'une application de robotique à travers une machine à états finis ou des algorithmes. Ce package représente la **partie Control**.
- Le package `RoboticCommunications` définit la communication entre les composants à travers des ports et des connecteurs. Ce package représente la **partie Communication**.
- Le package `Deployment` permet de spécifier les propriétés relatives au déploiement des composants en représentant les plateformes cibles (middleware ou

simultaurs) vers lesquelles le code des composants sera généré.

4.1.1.1 La partie Architecture Cette partie représente les concepts qui permettent de définir l'architecture d'un scénario de robotique sous forme d'un modèle à base de composants ayant des propriétés, des ports et des types. La partie Architecture comporte six packages comme le montre la figure IV.19 : le package *RoboticSystem*, le package *SystemEnvironment*, le package *DataType*, le package *RoboticMission*, le package *Platform* et le package *Deployment* que nous détaillerons dans ce qui suit.

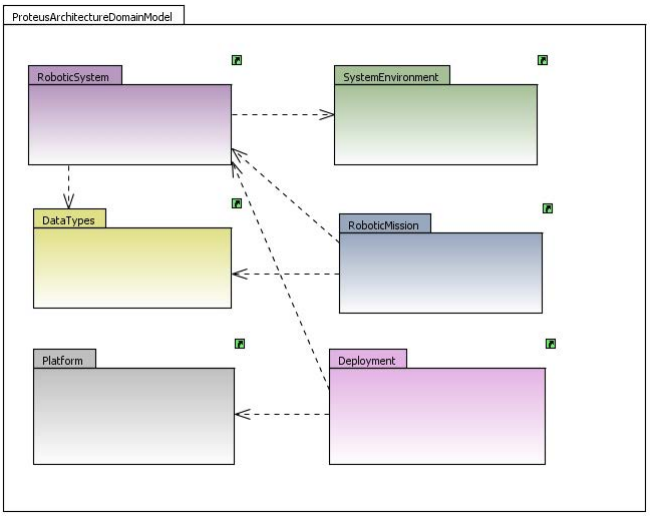

Figure IV.19 – RoboticArchitecture Package

1. **Le package *Robotic System*** (voir Figure IV.21). La méta-classe `System` représente un composant ayant une activité (attribut *activity*) et qui peut être local ou non (attribut *native*). Cet attribut permet d'indiquer si un composant qui satisfait le même rôle existe déjà localement sur la machine. Les attributs *libraryPath* et *LibraryComponentName* sont respectivement le chemin et le nom de ce composant existant qu'on peut associer au composant représenté. Un `System` a une ou plusieurs propriétés (`Property`), un ou plusieurs ports (`Port`) et des connecteurs (`Connector`) qui permettent de relier les ports entre eux. La méta-classe `Property` est soit une propriété d'un système dans le sens attribut (méta-classe `BasicProperty` de la figure IV.20) soit une partie d'un système (méta-classe `Part` de la figure IV.20).

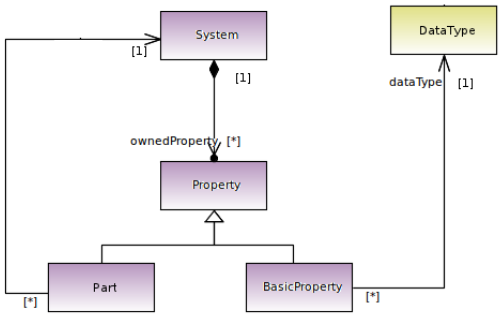

Figure IV.20 – La méta-classe Property

La méta-classe `Port` sera développée dans la partie Communication. Un `System` a un comportement exprimé à travers la méta-classe `Evolution Model` que nous détaillerons dans la partie Control. Un `RoboticSystem` est un `System` permettant de représenter un composant concret, il a une position et une orientation. La méta-classe `SensorSystem` représente les capteurs, l'attribut `frequency` est la fréquence de chargement des données des capteurs. La méta-classe `ActuatorSystem` représente les effecteurs. Ces deux méta-classes sont spécialisées dans d'autres types de capteurs et d'effecteurs spécifiques que nous ne présenterons pas car nous nous intéressons uniquement aux concepts du DSML et non aux détails spécifiques des capteurs et des effecteurs.

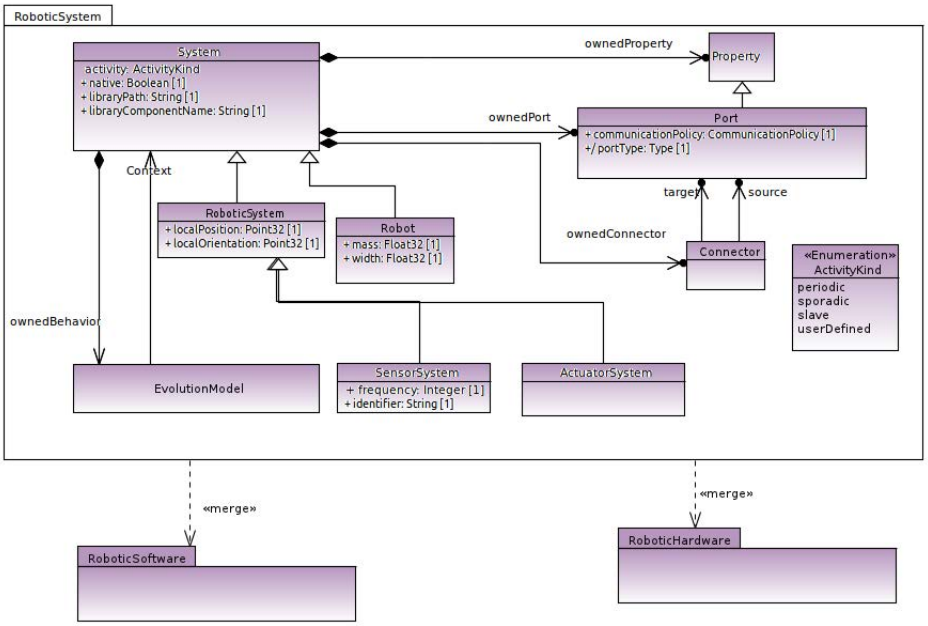

Figure IV.21 – Le package RoboticSystem

2. **Le package *SystemEnvironment***. Ce package définit les concepts de l'environnement où le robot évolue. Cette représentation permet de définir les environnements de simulation par exemple.
3. **Le package *DataType***. Ce package permet de spécifier les types de données qui vont être utilisés par les ports, les propriétés, etc. (voir figure IV.22).

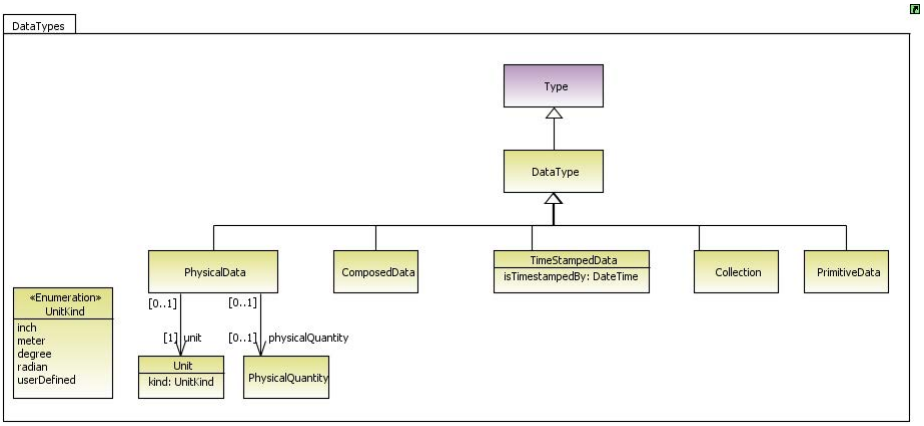

Figure IV.22 – RobotML Data Types

Un data type peut être :

– une donnée physique `Physical Data` : Ce type représente une unité mathématique associée à un objet physique. Cet object mathématique peut être considéré comme une **abstraction d'une donnée physique**. On trouve par exemple les types *Distance*, *Angle*, *Acceleration* comme le montre la figure IV.23. La sémantique exacte de ces types est gérée ensuite par l'utilisateur. En effet, au niveau du DSML, ces types n'ont pas de sémantique opérationnelle. Par exemple, le type distance peut désigner une distance par rapport à un obstacle ou une distance de sécurité, etc.

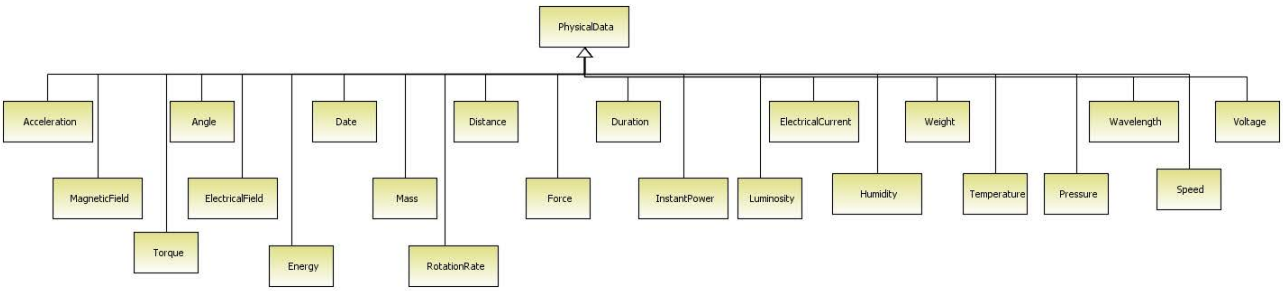

Figure IV.23 – physicalData

– un type composé `ComposedData`. C'est un type qui représente des données communes à des objets du domaine de la robotique. Comme par exemple `GPS`, `Image`, etc. (voir figure IV.24).

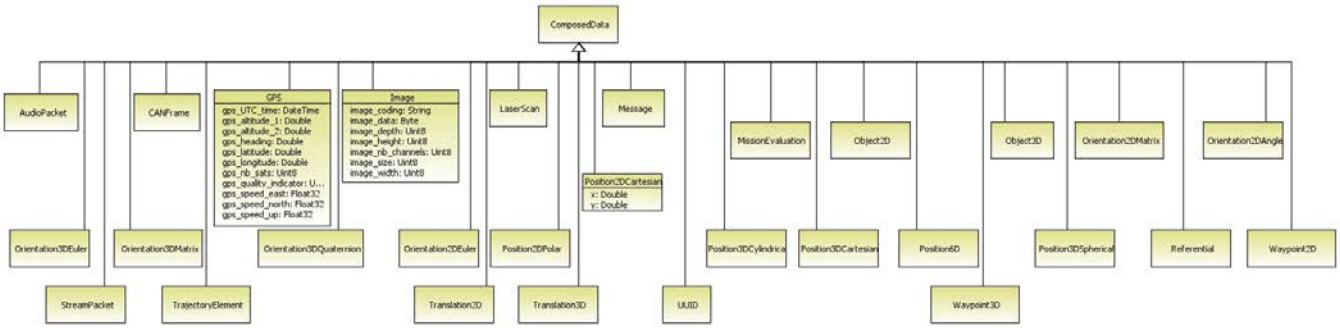

Figure IV.24 – Types composés de RobotML

– un type primitif `PrimitiveData` comme par exemple le type bool, int, char, etc. (voir figure IV.25).
– une collection c.-à-d. listes, tables de hashage, séquences, etc.
– un timestamped data qui associe un horodateur à une donnée pour connaître son évolution au fil du de l'exécution (si elle reste valide, etc.).

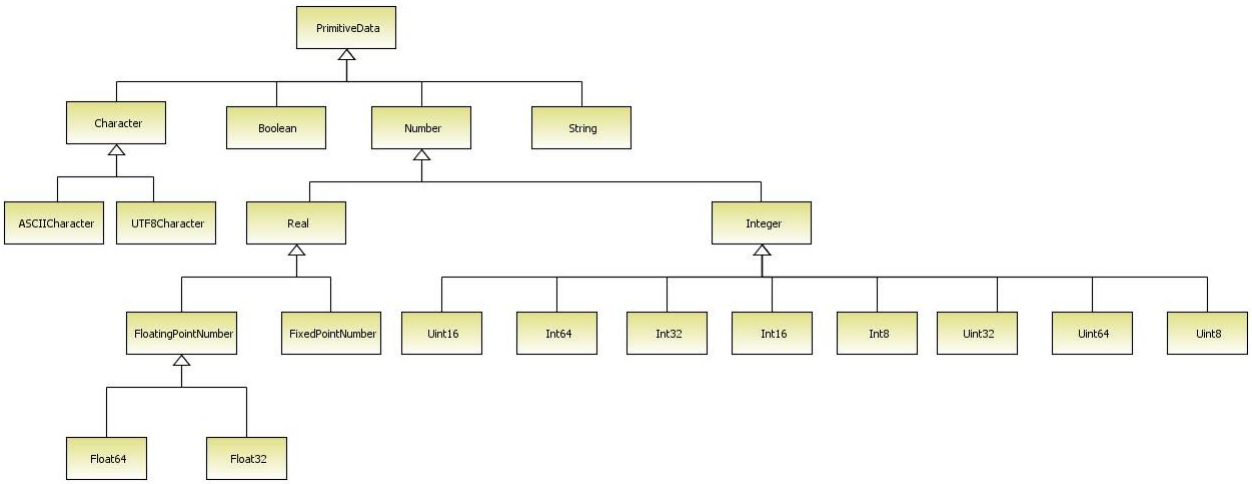

Figure IV.25 – Types primitifs de RobotML

Étant donné que le middleware ROS a été intégré à la plupart des plateformes cibles de RobotML (OROCOS, RT-Maps, etc.), nous avons également défini une librairie basée sur les types ROS afin de faciliter leur utilisation par les uti-

lisateurs de RobotML. Ces types sont représentés comme des `DataType` comme le montre la figure IV.26 où les geometry_msgs de ROS sont représentés.

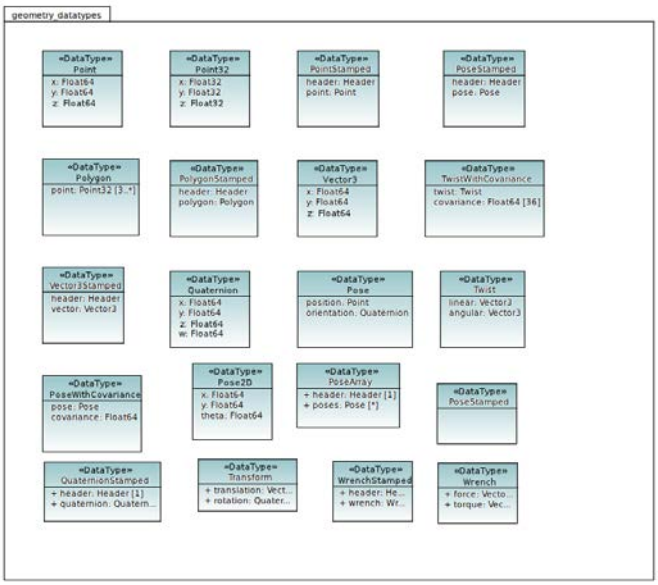

Figure IV.26 – Intégration des geometry_msgs de ROS à RobotML

4. **Le package *Robotic Mission***. Ce package décrit les concepts qui sont requis pour la définition d'une mission opérationnelle et qui sont utilisés par les composants de l'architecture qui réalisent cette mission. Ce package est typiquement utilisé pour la représentation des scénarios.

5. **Le package *Platform*** (voir figure IV.27). Il définit les plateformes qui représentent l'environnement d'exécution. La méta-classe `Platform` peut être un middleware (méta-classe `RoboticMiddleware`) ou un simulateur (`RoboticSimulator`). L'attribut *RoboticMiddlewareKind* est typpé par une énumération où les noms des middleware pris en charge par le DSML sont spécifiés. Aucune propriété des middleware n'est représentée. Le choix d'un middleware servira uniquement au déploiement. De même pour la méta-classe `RoboticSimulator` qui est spécialisée en trois méta-classes où il y a seulement besoin de spécifier la configuration souhaitée de l'exécution (temps-réel ou non, valeur de la gravité, etc.) selon le simulateur choisi.

6. **Le package *Deployment*** permet l'allocation d'un composant (méta-classe `System`) vers une plateforme (méta-classe `Platform`) comme le montre la figure IV.28.

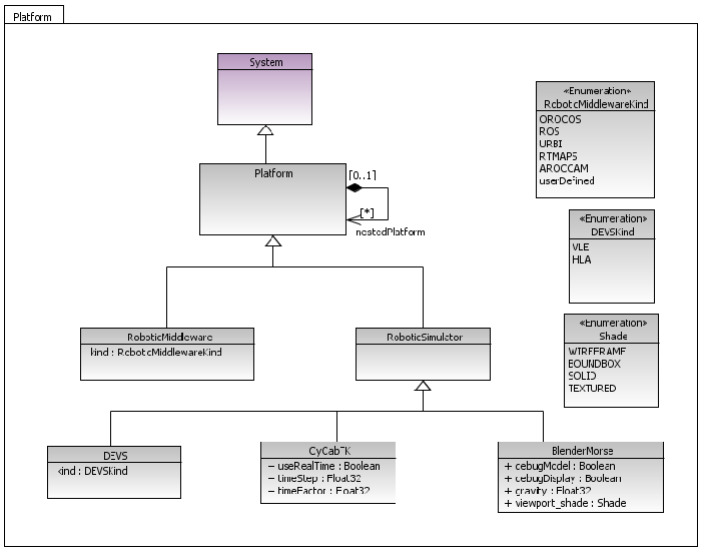

Figure IV.27 – Le package Platform

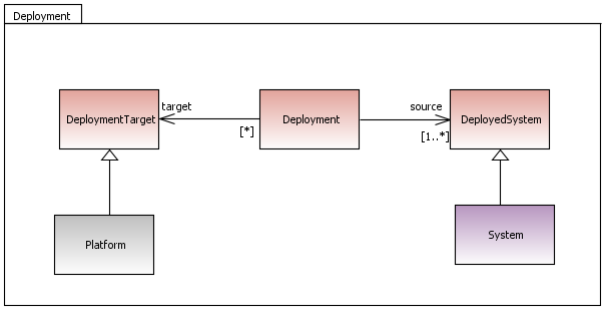

Figure IV.28 – Le package Deployment

Pour résumer, à partir de la partie architecture, il est possible de représenter le squelette d'un composant : ses ports, les types de ses ports ainsi que ses propriétés. Un composant peut être un composant matériel (capteur ou effecteur) ou un composant de contrôle. Dans la section suivante, nous présentons la partie control qui permet de spécifier le comportement d'un composant.

4.1.1.2 La partie Contrôle Cette partie du DSML est la partie responsable de la spécification du comportement des composants de contrôle d'un système de robotique. Nous allons maintenant expliquer quel contrôle nous souhaitons exprimer :

– Le contrôle à boucle fermée (Feedback control) : Dans ce cas le système doit avoir un retour (une connaissance) sur son état après l'exécution d'une commande.

– Le contrôle à boucle ouverte (open loop control) : Dans ce cas le système ne récupère pas les informations sur son état après l'exécution d'une commande.

On retrouve ces boucles de contrôle dans les architectures que nous avons présentées dans la section 3.3 du chapitre 1. Si l'architecture est réactive, le principe est celui de Stimulus/Réponse. Il y a donc une action qui est définie suite à un état dans lequel se trouve le robot. Si l'architecture est basée sur le comportement, il devrait être possible de suivre l'évolution des comportements du robot et de les activer/désactiver. Si l'architecture est hybride ou délibérative, le contrôle se base sur un plan. Il devrait alors être possible de représenter l'état du robot dans son plan.
Le contrôle peut alors être exprimé comme un automate qui permet l'expression des instructions d'un composant à un niveau plus abstrait que le code.
Pour commencer, l'utilisateur peut spécifier un automate où les états par lesquels le robot pourrait passer sont représentés. Il faudrait ensuite spécifier quels sont tous les états possibles à partir d'un état donné et sous quelle condition la transition entre ces états est déclenchée. Si le contrôle se base sur une boucle ouverte, la transition d'un état à un autre est déclenchée directement à partir du premier état. En revanche lorsqu'il s'agit d'une boucle fermée, la transition est déclenchée selon le retour de la commande exécutée.

La figure IV.29 montre une vue globale des packages qui interviennent pour la définition de la partie comportementale d'un composant. Le comportement des composants est défini sous forme de machines à états finis (FSM) ou avec des algorithmes.

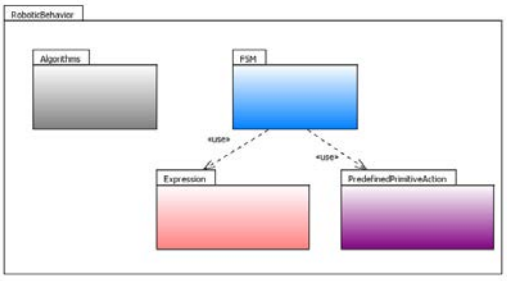

Figure IV.29 – Le package RoboticBehavior

La figure IV.30 donne une vue détaillée sur le package FSM. `Evolution Model`, conformément à sa description dans l'ontologie, est utilisé pour décrire le cycle de vie ou le comportement d'un composant (`System`). La méta-classe `FSM` hérite de

la méta-classe `EvolutionModel`. Une FSM est un automate composé d'états et de transitions. Ce package est fortement inspiré de la spécification des machines à états du méta-modèle d'UML.
Les états représentent les actions que le robot doit effectuer. Nous distinguons les états initiaux des étaux finaux dans les FSMs. L'état initial `Initial State` est le premier état visité lorsqu'une FSM est exécutée, il est typiquement utilisé pour effectuer les initialisations requises pour la FSM. L'état final `Final State` indique que l'exécution de la FSM est terminée. Les transitions (`Transition`) forment des liens entre les états de la FSM. Une transition a un état source *incoming* et un état destination *outgoing*. Une transition peut aussi avoir une `GuardCondition` et un évènement (`Event`). Les états et les transitions sont composés de `FSM Activities` spécifiant les actions à exécuter (méta-classe `Effect`) pendant un état ou lors d'une transition donnée.

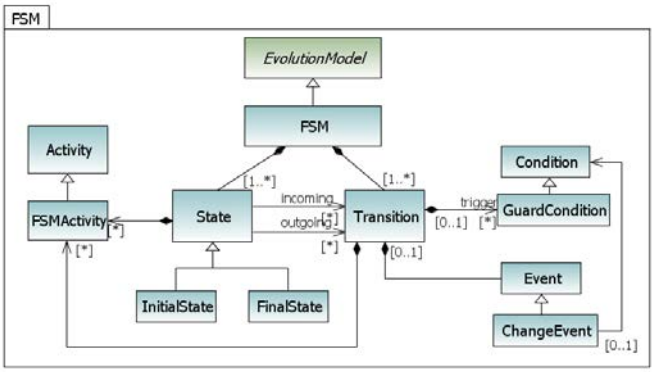

Figure IV.30 – Le package FSM

Dans la section suivante, nous présentons la communication entre les différents composants d'une application.

4.1.1.3 La partie communication

Cette partie doit permettre la spécification de la communication entre les composants à travers des ports et des connecteurs. Nous nous sommes inspirés des mécanismes de communication utilisés dans SmartSoft [82] et nous les avons adapté aux concepts décrits par l'ontologie.
La communication est réalisée à travers un échange synchrone ou asynchrone de données ou des appels de service. La figure IV.31 décrit les ports d'un composant. La méta-classe `Port` représente un point d'intéraction du système. On distingue deux sortes de ports :

– Les ports de données `DataFlowPort` qui permettent la communication à travers des flots de données entre les composants. La méta-classe `DataFlowPort` a pour type un `DataType` et possède un attribut *direction* = {In, Out, InOut} qui spécifie le sens de l'échange de données et un attribut *bufferSize* qui définit la taille du buffer où les données sont enregistrées. La direction des ports est l'un des exemples de l'union des concepts des plateformes cibles que nous avons évoqué lorsque nous avions parlé de l'extraction des abstractions pertinentes dans le DSML. En effet, la direction *InOut* n'est pas présente dans toutes les plateformes cibles. Cependant, interdire sa représentation au niveau du DSML engendrerait des contraintes pour l'utilisation des plateformes qui prennent en charge ce concept.
– Les ports de service `ServicePort` qui sont spécifiques à la communication client/serveur. La méta-classe `ServicePort` a un attribut *serviceKind* qui indique si le port est fourni ou requis. Un `ServicePort` a pour type une interface qui définit un ensemble d'opérations abstraites.

La méta-classe `CommunicationPolicy` spécifie le type de communication utilisé (attribut synchronizationPolicy= {Synchrone/Asynchrone}) et la stratégie employée par le buffer (attribut *strategyPolicy*= {FIFO, LIFO}).

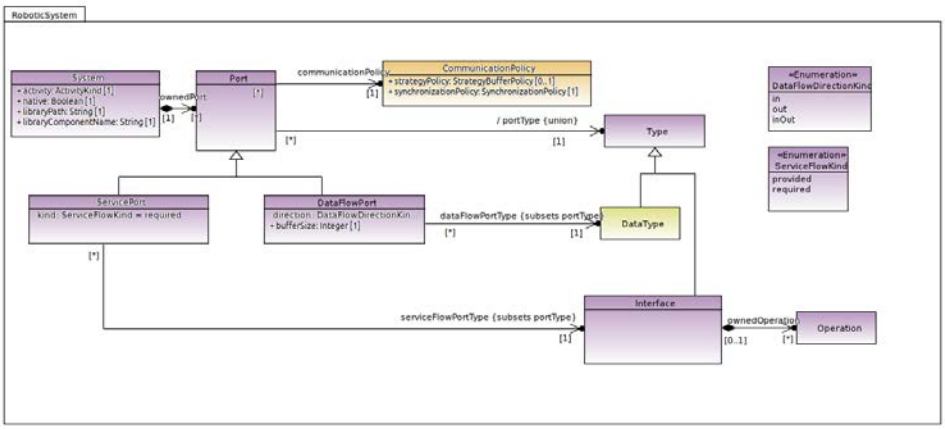

Figure IV.31 – Les mécanismes de communication

4.1.2 Le profil RobotML

Le profil RobotML définit des extensions des méta-classes du méta-modèle UML afin de les adapter aux concepts identifiés dans le modèle de domaine RobotML.

L'objectif de notre démarche est de pouvoir définir un éditeur graphique avec Papyrus [61] qui se base sur le profil RobotML et qui permet de manipuler les concepts identifiés à travers des modèles de scénarios de robotique.
Pour définir le profil, les abstractions identifiées dans les parties représentées dans le modèle de domaine RobotML ont été reprises pour étendre les méta-classes d'UML. Le profil comporte alors trois parties principales conformément au modèle de domaine RobotML : architecture, communication et contrôle.
Dans la partie architecture du profil, la méta-classe `System` et toutes les méta-classes qui la spécialisent sont transformées en stéréotypes qui étendent la méta-classe `Class` du méta-modèle UML comme le montre la figure IV.32. Les propriétés des méta-classes de RobotML comme par exemple les méta-classes `RoboticSystem`, `Robot` et `Software` sont transformées en *tagged values* des stéréotypes `RoboticSystem`, `Robot` et `Software` du profil.

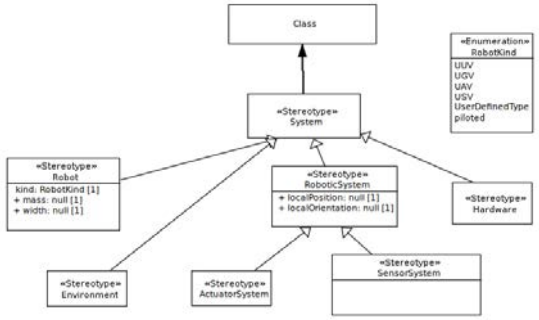

Figure IV.32 – Extrait de la partie Architecture : Stéréotypes extensions de la méta-classe `Class` d'UML

Concernant la partie communication, la méta-classe `Port` du modèle de domaine RobotML et toutes les classes qui la spécialisent sont également transformées en stéréotypes qui étendent la méta-classe `Port` du méta-modèle UML comme le montre la figure IV.33. Les propriétés des méta-classes `ServicePort` et `DataFlowPort` du modèle de domaine RobotML sont définies comme des *tagged values* des stéréotypes `ServicePort` et `DataFlowPort` du profil.

De la même façon, les méta-classes de RobotML représentant les aspects qui relèvent du contrôle ont été représentés à travers des stéréotypes qui étendent les méta-classes définissant les machines à états dans le méta-modèle UML (`State`, `Transition`, etc.) comme le montre la figure IV.34 pour les stéréotypes `State` et `Transition`. Concernant la méta-classe `Algorithm` de RobotML, elle est définie comme un stéréotype qui étend la méta-classe `Operation` du méta-modèle UML.

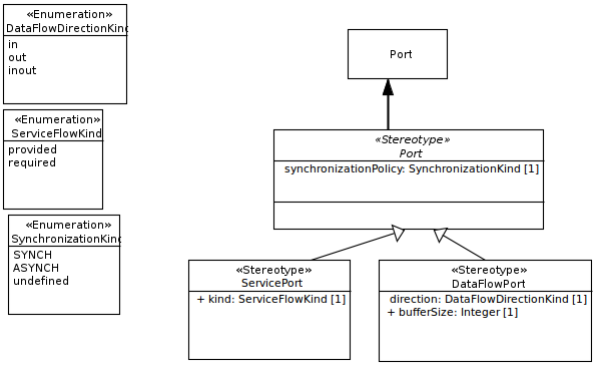

Figure IV.33 – Extrait de la partie communication : Stéréotypes extensions de la méta-classe `Port` d'UML

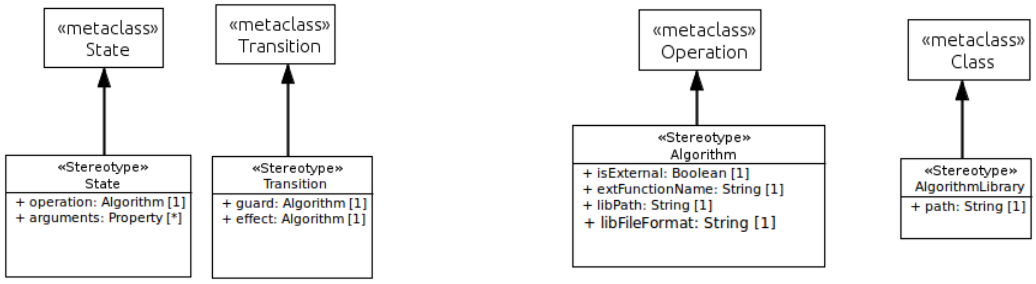

Figure IV.34 – Extrait de la partie Control : Stéréotypes extensions des méta-classes du méta-modèle d'UML

Chaque stéréotype du profil que nous défini peut avoir une icône associée pour sa représentation graphique. Nous allons présenter dans ce qui suit l'éditeur graphique de RobotML.

4.2 Syntaxe concrète : L'éditeur graphique de RobotML

L'éditeur graphique de RobotML a été conçu en utilisant l'outil Papyrus [61]. Afin de définir l'éditeur graphique, des icônes peuvent être rattachées graphiquement aux stéréotypes du profil de RobotML. Selon la sémantique d'utilisation l'une des icônes est affichée.

La partie gauche de la figure IV.35 montre les icônes attachées aux stéréotypes `SensorSystem`, `ActuatorSystem`, `DataFlowPort` et `ServicePort`. Ces icônes sont ensuite représentées dans la palette (partie droite de la figure). Les stéréotypes `SensorSystem`, `ActuatorSystem` ont une seule icône attachée qui représente un capteur ou un effecteur. Le stéréotype `DataFlowPort` a trois icônes attachées, chacune

correspond à un type de DataFlowPort selon la direction utilisée {In, Out, InOut}. Le stéréotype `ServicePort` n'a que deux états possibles {Required, Provided}, chacun d'entre eux a une icône qui lui correspond.

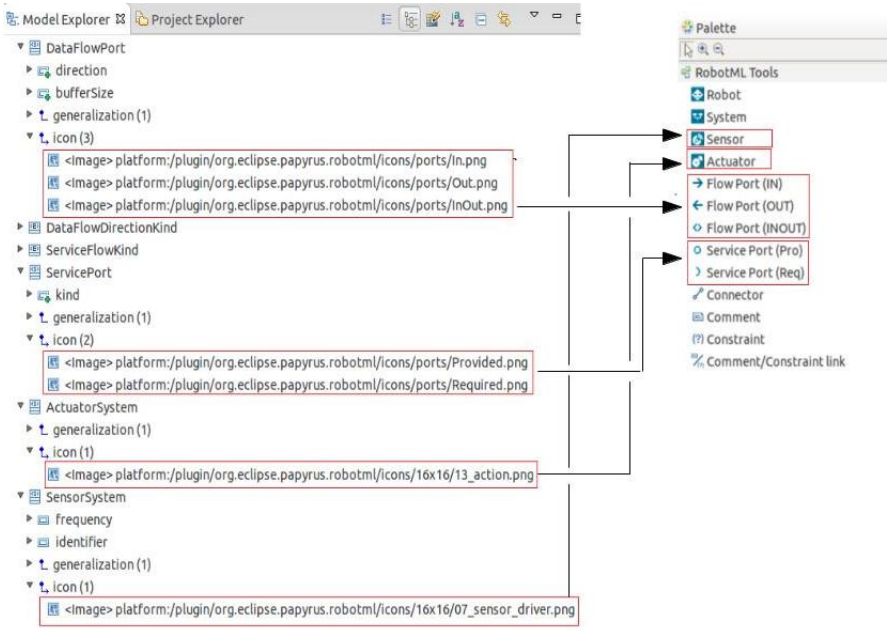

Figure IV.35 – À gauche, les icônes attachées aux stéréotypes, à droite la palette montrant les différentes icônes

En plus des icônes, il est possible d'associer un ensemble de stéréotypes à des diagrammes dédiés à une représentation d'une partie du modèle. Plusieurs diagrammes sont proposés comme le montre la figure IV.36. Le model Explorer contient les éléments du modèle représentés à travers les différents diagrammes. Le diagramme de définition des composants permet de représenter les composants d'une application ainsi que les capteurs et les effecteurs. Le diagramme de machine à états et la palette qui lui est associé contient les concepts State, Final State, Initial State et Transition. La vue Properties permet de spécifier les propriétés des éléments du modèle.

À partir des modèles représentés avec cet éditeur graphique on peut générer du code exécutable vers une ou plusieurs plateformes cibles. La section suivante présente la génération de code vers le middleware OROCOS.

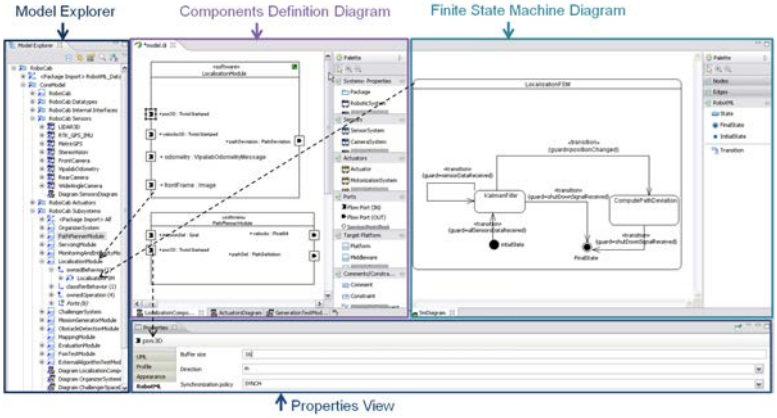

Figure IV.36 – Environnement de modélisation de RobotML

5 Intégration du middleware OROCOS : génération de code

Les concepts de RobotML étant définis à partir de l'état de l'art sur les middlewares et les simulateurs et d'une ontologie de la robotique, regroupent l'union des concepts identifiés dans les plateformes cibles ainsi que des abstractions du domaine. Par exemple, les ports d'entrée sortie présentés précédemment : le concept *InOutPort* n'est pas présent dans toutes les plateformes. Nous l'avons représenté dans le modèle du domaine car nous avons fait le choix de ne pas imposer de restrictions sur les concepts des plateformes. Ainsi il est possible de représenter les plateformes qui utilisent ce concept.

La génération de code permet de founir à partir d'un modèle de scénario une application exécutable facilitant ainsi le développement des applications de robotique. Les plateformes cibles des générateurs peuvent évoluer mais les abstractions que nous avons utilisées sont suffisamment stables pour supporter cette évolution à court terme.

En d'autres termes, le développement des générateurs de code définis à partir de RobotML est un processus plus rapide que l'évolution des plateformes cibles. Bruyninckx et al. [3] ont argumenté le fait qu'ils ne pouvaient pas intégrer leur approche de la séparation des responsabilités à leur méta-modèle (voir figure III.8) par le fait que les plateformes cibles se basaient sur des architectures stables qui ne vont pas évoluer vers d'autres architectures.

Ainsi, la définition des générateurs de code faciliterait le développement des appli-

cations de robotique et permettrait d'accélérer le processus de développement.

La première étape pour l'intégration des plateformes cibles est d'établir des règles de correspondance entre les éléments du DSML et les plateformes cibles.
Grâce au respect de RobotML des exigences du domaine et de celles des middleware et des simulateurs de robotique, il est possible d'établir des règles de transformations à partir de RobotML vers plusieurs plateformes hétérogènes.
Dans le cadre du projet PROTEUS, plusieurs générateurs de code ont été réalisés vers : OROCOS-RTT[47], RTMaps[113], Urbi [114] et Arrocam [115] et les simulateurs MORSE [116] et CycabTK [117]. Notre approche se base sur les techniques de la MDA présentées précédemment à l'exception que notre PSM est le code. Nous utilisons une API commune implantée qui permet d'interroger les éléments du modèle. En se basant sur cette API commune, chaque générateur définit les règles de transformation vers sa plateforme cible en utilisant l'outil Acceleo [67].

Le middleware ROS a joué un rôle important dans le développement des générateurs de code. Étant donné qu'il a été intégré aux plateformes cibles prises en charge par RobotML, nous utilisons les topics ROS pour l'échange des données entre les composants générés vers des plateformes hétérogènes.

Nous présentons dans cette section le générateur de code OROCOS-RTT qui définit des transformations de modèle vers texte à partir de RobotML. Le générateur de code a été implanté en utilisant *Acceleo*.
OROCOS (Open RObot COntrol Software) est un framework temps-réel à base de composants C++ [47], appelé aussi OROCOS-RTT (Real-Time Toolkit) pour son aspect temps-réel. Les composants OROCOS intéragissent par un échange de données, d'évènements ou de services à travers des ports. L'échange de données est effectué par des *data flow* ports d'entrée/sortie, les événements indiquent un changement dans le système et l'échange de services est effectué à travers des simples appels de méthodes. Les composants OROCOS peuvent être définis sous forme de machine à états finis hiérarchiques.

5.1 Génération du squelette d'un composant

Le tableau IV.6 montre la correspondance entre les concepts du DSL architecture décrivant les concepts relavant de la structure globale d'un composant et les

Concepts de RobotML	concepts d'OROCOS
Property	Task Context
Attribute	Attribute
Port	–
Primitive DataType	C++ types
Composed DataType	Data structure
Interface	Abstract class
Attribute	Attribute
Interface Operation	Virtual Method
Operation	Operation
Parameter	Parameter

Table IV.6 – Correspondance entre le DSL Architecture et les concepts d'OROCOS

concepts d'OROCOS.
Les composants OROCOS héritent de l'interface `TaskContext` et possèdent des propriétés et des threads d'exécution. Concernant les interfaces, elles sont générées comme des classes abstraites ayant des méthodes virtuelles. Ces méthodes sont Concernant les Data Types, les `Primitive DataTypes` sont transformés en types C++. Concernant les types `ComposedDataTypes` ils sont générés comme des structures de données.

5.2 Génération de la communication entre les composants

Concepts de RobotML	concepts d'OROCOS
Data Flow Port direction {In}	Input Port
Data Flow Port direction {Out}	Output Port
Service Port {Provided }	this->provides(Operation)
Service Port {Required}	this->requires(OperationCaller)

Table IV.7 – Correspondance entre le DSL Communication et les concepts d'OROCOS

Les composants échangent les données et les appels de service à travers des ports de données (`DataFlowPort`) et des ports de service (`ServicePort`). Selon leurs directions, les ports de données sont transformés en *InputPort* ou en *OutputPort* comme le montre le tableau IV.7. Les ports de service, quant à eux, sont transformés en *ServicePort* qui fournissent ou requièrent une opération. Si le service est requis, le composant fait appel à l'opération définie dans l'interface à travers le mot clé *OperationCaller* en précisant le type de l'opération et ses paramètres. Dans le cas contraire (c.-à-d. si le service est fourni), le composant implémente l'opération en

question et indique qu'il la fournit en utilisant la méthode *provides*.

L'échange de données à travers les composants est effectué à travers des ports de données. RobotML fournit un ensemble de types communément utilisés en robotique (`Composed DataType`) sous forme de bibliothèques. Ces bibliothèques permettent de réutiliser ces types sans avoir à les redéfinir à chaque fois. Selon la nature de ces composants et le plan de déploiement, les types de ces ports sont générés :

- Si le composant est un composant de contrôle (`System`) et est connecté à d'autres composants de contrôle déployés sous le même middleware alors les types de ses ports sont les types indiqués dans le modèle.
- Si le composant est un composant de contrôle et est connecté à un autre composant déployé sous un autre middleware alors le type de ses ports est traduit en type ROS.
- Si le composant est un composant matériel (c.-à-d. capteur ou effecteur) alors les types de ses ports seront traduits en types ROS.
- Si le composant est composant de contrôle et est connecté à un composant matériel alors les types de ses ports seront traduits en types ROS.

Les bibliothèques définissant les types utilisés par les propriétés ou les ports d'un composant sont automatiquement importées dans le fichier source du composant.

La connexion entre les composants est configurée dans un script OPS (OROCOS Program Script) qui permet la spécification de la périodicité, du type de communication entre les composants (ROS ou CORBA) comme le montre le tableau IV.8. Le plan de déploiement intervient également pour cette configuration. Une distinction est effectuée entre la communication inter-composants du même middleware (mot clé *connectPeers*) pour indiquer que la communication va s'appuyer sur CORBA. Autrement, le mot clé *stream* est utilisé pour spécifier que la connexion se fait entre des composants de natures différentes et que ROS sera utilisé.

5.3 Génération du comportement d'un composant

Chaque composant a un comportement décrit en utilisant une machine à états finis (FSM) ou un algorithme. Les FSMs peuvent être définies grâce au package rFSM [2] intégré à OROCOS. Il permet la définition de scripts en Lua [118] puis de

2. http://people.mech.kuleuven.be/~mklotzbucher/rfsm/README.html#sec-1

Concepts de RobotML	Fichier OPS
Communication Policy	setActivity("ComponentA", period, priority, scheduler) setActivity("ComponentB", period, priority, scheduler) loadComponent("ComponentA", "A") loadComponent("ComponentB", "B")

Connector
(composants homogènes)
	var connPolicy x ;
	x.type = DATA ; //DATA = 0, BUFFER = 1
	x.size = 1 ; //size of the buffer
	x.lock_policy = LOCK_FREE ; //UNSYNC = 0, LOCKED = 1, LOCK_FREE = 2
	x.transport = 3 ; //ROS = 3
	stream("componentA", x)

Connector (composants hétérogènes)	connectPeers("ComponentA", "ComponentB")

Table IV.8 – Correspondance entre le DSL Communication et les concepts d'OROCOS

les charger par les composants OROCOS. La tableau IV.9 décrit la correspondance entre les concepts de RTT-Lua et les concepts de RobotML.

Concepts de RobotML	concepts d'OROCOS
State Machine	StateMachine
Initial State	initial state{ entry{} run{} exit{}}
Final State	final state{ entry{} run{} exit{}}
State	state{ entry{} run{} exit{}}
Transition Event Guard Effect	rfsm.transition {src="InitialState", tgt="state1" ,events{} ,guard=function() <body> end ,effect=function() <body> end }

Table IV.9 – Correspondance entre le DSL Contrôle et les concepts d'OROCOS

La correspondance entre les états est triviale. Les gardes et les effets sur les transitions sont des fonctions dont le corps est défini par l'utilisateur à travers un stéréotype d'UML appelé `OpaqueBehavior`. À la génération, ces fonctions sont capturées à partir du modèle et injectées à la place de la balise *<body>*. Un exemple de requête permettant de récupérer les transitions définies dans une machine à

états du modèle est donné dans le listing IV.6. Cette requête retourne un ensemble de transition (*sequence(Transition)* et fait appel à la fonction *getTransitions(org.eclipse.uml2.uml.StateMachine)* du template Acceleo `FSMQueries`.

```
[query public getTransitions(sm : StateMachine) : Sequence(Transition) =
        invoke('org.eclipse.robotml.generators.acceleo.mmqueries.FSMQueries',
        'getTransitions(org.eclipse.uml2.uml.StateMachine)', Sequence{sm})
/]
```

Listing IV.6 – Requête pour récupérer les transitions d'une machine à états

D'autres requêtes sont également utilisées pour générer le code d'une machine à états. Un extrait d'un tempate Acceleo définissant les règles de transformation pour chaque requête sur les transitions, leurs gardes et leurs effets sont présentées dans listing IV.7. Pour chaque transition on récupère à partir des triggers qui lui sont associés les guards et les opérations qui leur sont associés (lignes 13 - 23) ainsi que les effets (lignes 24 - 34).

```
1  [let transitions : Sequence(RobotML::Transition) = getTransitions(fsm)]
2    [for(transition : RobotML::Transition | transitions)]
3       rfsm.transition {src="
4  ␣␣␣␣␣[transition.base_Transition.source.name/]",
5       tgt="[transition.base_Transition.target.name/]"
6       [if(hasTriggers(transition.base_Transition))]
7        , events={
8       [for (trig: Trigger| transition.base_Transition.trigger)]
9       '[trig.name/]'
10      [/for]
11      }
12      [/if]
13      [if(transition.guard <> null)]
14      [let guard : RobotML::Algorithm = transition.guard]
15      ,guard=function()
16      [let op : Operation = getAssociatedOperation(guard.base_Operation)]
17         [for(o : OpaqueBehavior | getOperationMethod(op))]
18             [o._body/]
19         [/for]
20      [/let]
21      end
22      [/let]
23      [/if]
24      [if(transition.effect <> null)]
25      [let effect : RobotML::Algorithm = transition.effect]
26         ,effect=function()
27         [let op : Operation = getAssociatedOperation(effect.base_Operation)]
28           [for(o : OpaqueBehavior | getOperationMethod(op))]
29               [o._body/]
30           [/for]
31         [/let]
32         end
33     [/let]
34     [/if]
35     },
36     [/for]
37 [/let]
```

Listing IV.7 – Template de génération des transitions des machines à états vers OROCOS

Nous avons implanté des templates définissant des règles de transformation pour tous les éléments du modèle vers du code OROCOS.
Le but du générateur OROCOS consiste à fournir une application compilable et exécutable. Pour ce faire, un ensemble d'artefacts est également généré notamment :

- un fichier makefile
- un fichier CMakeLists.txt spécifiant le chemin des composants à charger pour l'application ainsi leurs noms.
- un fichier manifest.xml spécifiant les librairies à importer pour l'application.

Pour résumer, le générateur de code OROCOS prend en entrée le modèle représenté avec les outils de RobotML et fournit un ensemble de fichiers correspondants aux concepts de notre DSML comme le montre la figure IV.37.

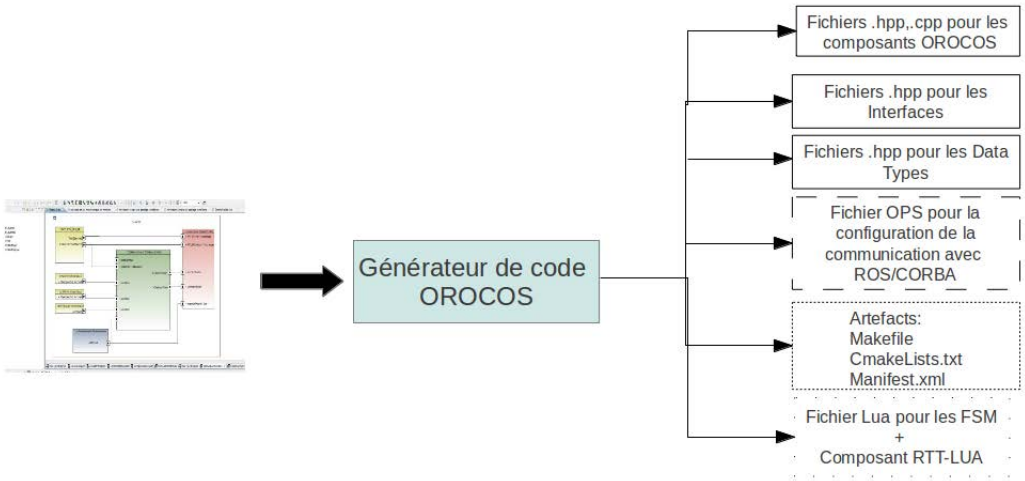

Figure IV.37 – Génération de code avec OROCOS

6 Utilisation : chaîne d'outillage RobotML

Afin de valider notre DSML sur des exemples industriels, plusieurs scénarios (appelées challenges) ont été proposés dans le cadre du projet PROTEUS. RobotML fournit une chaîne d'outillage pour le développement des applications de robotique allant de la modélisation jusqu'à la génération de code et le déploiement vers des simulateurs ou des robots réels.
L'utilisateur de RobotML commence par modéliser sa mission en représentant les types de données et les interfaces, les composants de contrôle et matériels, ainsi que

les connexions entre eux.
Une fois le modèle validé, l'utilisateur définit un plan de déploiement qui consiste en un ensemble de middlewares et/ou de simulateurs inter-connectés pour former une plateforme d'exécution pour l'application finale. Chacun des composants définis doit être alloué vers une plateforme cible d'un modèle de déploiement. Typiquement, le système de contrôle du robot modélisé est généré vers un ou plusieurs middleware (selon le plan de déploiement défini). Quant aux capteurs, aux actionneurs et à l'environnement du robot, ils sont générés vers le simulateur choisi. De plus, l'utilisateur a la possibilité de réutiliser des composants déjà existants sur la plateforme qu'il souhaite en spécifiant leur chemin et leur nom. Dans ce cas, pour ce composant, seule sa connexion avec le reste des composants est générée. L'utilisateur peut maintenant générer du code exécutable à partir du plan de déploiement qu'il a défini.

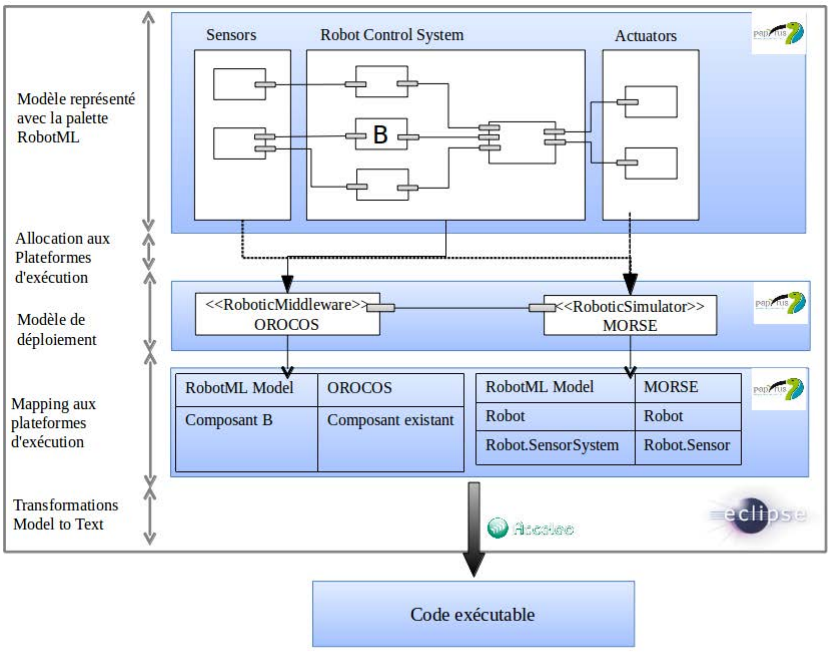

Figure IV.38 – Etapes de modélisation en utilisant RobotML

7 Validation et Expérimentations

Pour réaliser la validation de RobotML et de la génération de code vers OROCOS, nous avons choisi d'utiliser l'un des scénarios proposés dans le cadre du projet

PROTEUS par l'ONERA [3] appelé le scénario aérien (AIR OARP).

7.1 Scénario de l'AIROARP

Le scénario Air OARP considère le problème de la planification pour la perception et la perception pour la planification d'un véhicule aérien sans pilote. Le sujet de recherche adressé par Air OARP concerne les questions en matière de planification de mouvement pour la recherche et le suivi d'une cible. La plateforme ReSSac sera utilisée afin de réaliser les différentes expérimentations de site Caylus. Le site Caylus est une zone rurale et légèrement urbanisée. La zone comprend des routes, des pistes, des arbres et des bâtiments. Certains bâtiments comme les granges et les hangars ont de grandes portes ouvertes. Certains intrus peuvent tenter de traverser la zone, de se cacher dans cette zone ou tenter de s'échapper quand ils sont détectés. L'objectif pourrait être alors de détecter, reconnaître, identifier, localiser et suivre les intrus. La surveillance d'une zone est incluse dans un carré avec 400 mètres de long bord et est réalisée en utilisant un véhicule aérien sans pilote (UAV). Quand un intrus est reconnu dans le domaine d'un appareil photo, l'objectif peut être de le garder dans le domaine aussi longtemps que possible. Dans la section suivante, nous montrons la modélisation de ce scénario avec RobotML.

7.2 Conception du scénario avec RobotML

Nous commençons tout d'abord par présenter les différents types de données nécessaires pour ce scénario (voir figure IV.39). Le type `State` est un type utilisateur qui est défini par un numéro de phase, un compteur de trames, une altitude en baromètre, la vélocité et d'autres attributs qui spécifient l'état dans lequel se trouve le robot.
Le type `Image` est un type utilisateur qui désigne l'image d'une caméra. Il est défini par une largeur, une longueur, un nombre de pixels et un attribut qui indique si cette image a une couleur ou non.
Le type `Waypoint` est un type utilisateur qui indique les propriétés d'un point cible. Ce point a des coordonnées en 3 dimensions (*m_x, m_y, m_z*). Ce type permet aussi d'indiquer la direction que doit prendre le robot pour atteindre ce point ainsi que la vitesse d'approche pour aller à ce point.

3. http://www.onera.fr/

Le type `Way` est un type qui spécifie le chemin du robot. Enfin, les types primitifs `Double`, `bool`, `unsigned` et `char` sont représentés.

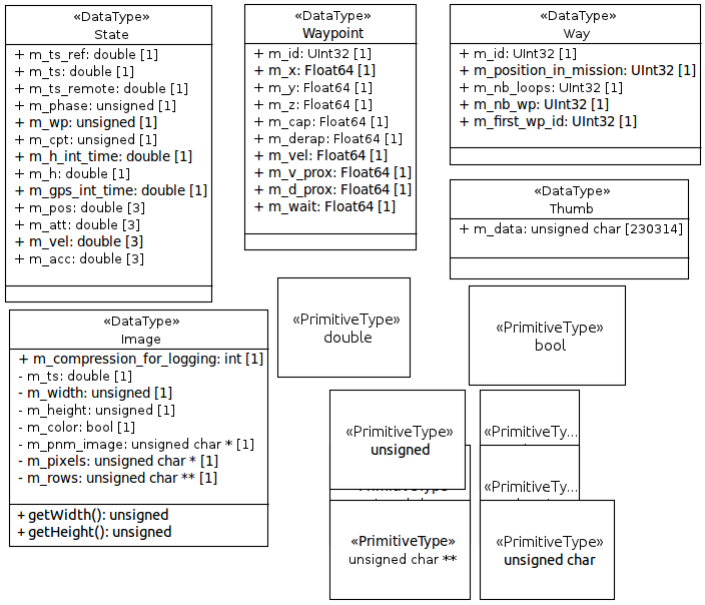

Figure IV.39 – Modèle des types de données AIR OARP

Après les types des données, les interfaces ont été représentées avec RobotML (voir figure IV.40). L'interface `ObcInterface` est l'interface qui définit les actions du robot (par exemple : *move*, *goTo*, *track* et l'évolution de sa trajectoire dans son environnement (par exemple *wayPointadd*, *wayPush*, *obstacleAdd*). L'interface `CameraInterface` définit les opérations relatives aux images perçues par la caméra du robot (par exemple *setBrighteness*, *setGain*, etc.).

Après avoir défini les types et les interfaces qui définissent les services que les composants vont s'échanger, les composants de l'application sont maintenant représentés. Le composant `Obc` du package *AvionicSystem* prend en entrée les informations des capteurs de localisation afin de récupérer les informations sur l'état du robot en implémentant les opérations de l'interface `ObcInterface`.

Le composant `m_talc_cam_driv` (voir figure IV.42) est un driver qui transforme les données physiques envoyées par la caméra en données de type Image (le type utilisateur défini). Ce composant implémente aussi les opérations de l'interface `CameraInterface`.

Ces deux composants envoient ensuite l'état du robot et l'image perçue au composant `RMaxControlSystem` qui va décider de l'action à effectuer. Les opérations des

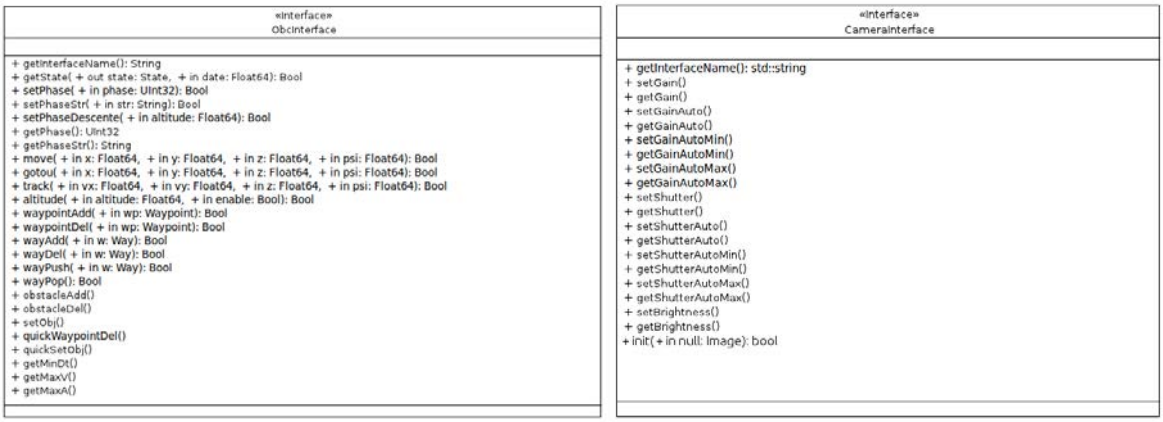

Figure IV.40 – Les interfaces du scénario AIR OARP

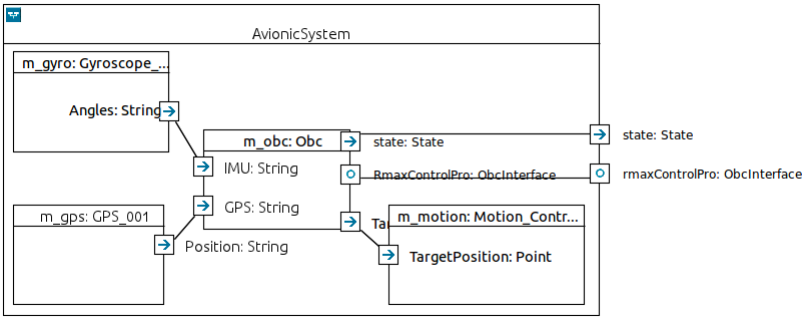

Figure IV.41 – Le composant AvionicSystem

deux interfaces `ObcInterface` et `CameraInterface` sont des services requis par le composant de contrôle principal `RMaxControlSystem`.
Dans la section suivante, nous présenterons la génération de code vers le middleware OROCOS.

7.3 Génération de code vers OROCOS

La génération de code comme nous l'avons présentée précédemment permet de générer pour chaque concept de RobotML le code associé. Les interfaces et les types de données ont été traduits en structures de données définissant leurs propriétés et leurs opérations. Les composants représentés ont été traduits en composants OROCOS. Les interactions entre eux sont tranduites par des propriétés au niveau des fichiers de configuration ainsi qu'au niveau des ports.
Enfin pour le composant de contrôle `RMaxControlSystem`, nous avons généré une

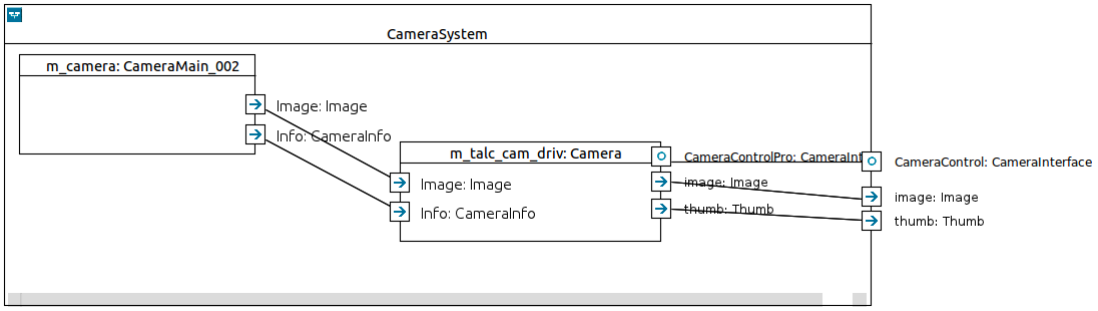

Figure IV.42 – Le composant CameraSystem

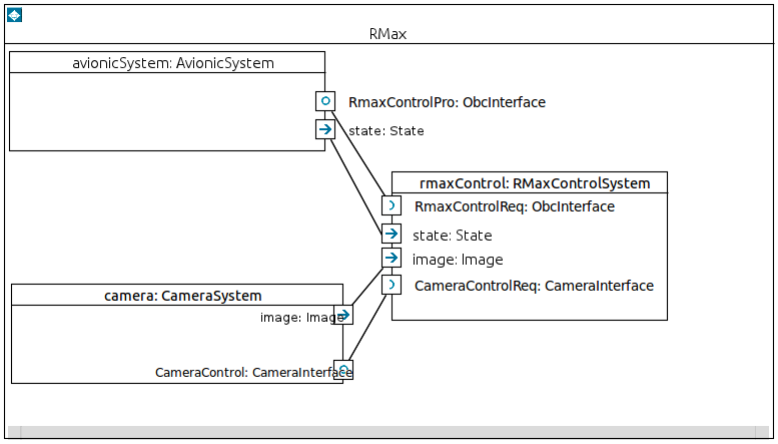

Figure IV.43 – Modèle du scénario AIR OARP

machine à états finis. Les résultats de cette génération de code sont disponibles sur la page [4].

8 Discussion

L'objectif principal de RobotML est de fournir aux roboticiens un langage dédié qui facilite le développement de leurs applications et qui répond aux exigences du domaine. Dans ce contexte, un ensemble d'exigences a été défini dans 4.1. RobotML a apporté une réponse à ces exigences :

1. **Facilité d'utilisation.** Grâce à RobotML, les utilisateurs du DSL peuvent modéliser leurs applications sans avoir à connaître ou à maîtriser les lanagages

4. https://github.com/RobotML/RobotML-documentation/blob/master/RobotMLCodeGenerators/OROCOSGenerator/index.rst

de programmation des plateformes de robotique. En effet, les détails des plateformes sont masqués pour les utilisateurs du DSL et les concepts utilisés dans RobotML sont basés sur une ontologie définie par des experts de la robotique.

2. **Spécification d'une architecture à base de composants.** L'architecture utilisée est une architecture à base de composants de contrôle et matériels.
3. **Spécification des comportements des composants**. RobotML permet la spécification du comportement des composants à travers des machines à états finis ou des algorithmes.
4. **Neutralité vis à vis des architectures de robotique.** La plupart des plateformes de robotique sont basées sur une architecture à base de composants. C'est pour cette raison que RobotML n'est spécifique à aucune autre architecture (délibérative, réactive ou hybride). N'importe quelle architecture peut donc être définie avec RobotML.
5. **Plusieurs plateformes cibles hétérogènes.** En raison de la distinction de la nature des composants au niveau de la modélisation (matériel ou contrôle) et de la possibilité de spécifier un plan de déploiement, il est possible de générer du code vers plusieurs plateformes hétérogènes. Lorsqu'il s'agit de faire interopérer ces plateformes hétérogènes (c.-à-d. : un middleware et un simulateur ou plusieurs middlewares) la communication s'appuie sur les messages ROS (automatiquement pris en charge par les composants générés) en raison de son intégration dans la plupart des plateformes de robotique actuelles.
6. **Indépendance vis à vis des plateformes cibles.** Grâce à l'ontologie du domaine, les concepts de RobotML sont abstraits et se basent sur la terminologie du domaine. Cependant, ces concepts correspondent aussi aux plateformes cibles sans être spécifiques à une plateforme particulière. Il n'y a pas de restriction imposée par une plateforme.
7. **Prise en charge de l'évolution des plateformes** Etant donné que le DSML est indépendant des détails des plateformes cibles, l'évolution des plateformes cibles n'affecte pas le modèle de domaine RobotML. Cette évolution pourrait cependant avoir un impact sur les générateurs. Notons quand même que le temps de développement des générateurs est moins important que l'évolution des plateformes cibles).

8. **Evolution du DSL.** RobotML est défini de manière itérative à partir d'exemples exécutables. Cette approche a été combinée avec une approche complémentaire *top-down* à partir d'une ontologie du domaine de la robotique définissant des concepts précis. L'évolution du DSL est considérée comme un processus intégré et incrémental au processus du développement. Dans certains cas, une absence de correspondance entre le modèle de domaine et les plateformes cibles a été constaté. Dans ce cas, ce concept est ajouté au DSL et des nouvelles règles de transformation sont définies pour la génération de code. Par exemple, nous avons constaté que l'utilisateur doit être capable de spécifier la taille du buffer pour l'échange de données entre les composants au niveau de la modélisation. Un nouvel attribut a alors été ajouté à la méta-classe `CommunicationPolicy` de RobotML.
9. **Abstraction du matériel** Les types `Physical Data` permettent de définir des abstractions à partir du matériel. Cependant, ils représentent des abstractions générales et n'ayant pas de sémantique précise. C'est pour cette raison qu'ils ont été définis au niveau du modèle du domaine et pas implémentés dans la syntaxe concrète de RobotML.

La validation de RobotML et du générateur de code OROCOS a été effectuée sur le scénario AIR OARP que nous présenterons dans le chapitre suivant.

9 Conclusion

Nous avons présenté dans ce chapitre notre DSML pour la robotique : RobotML. Les concepts de RobotML sont des concepts du domaine définis par des experts en robotique. Nous avons montré que ces concepts nous ont permis de représenter des scénarios de robotique indépendamment des plateformes d'exécution (middleware et simulateurs). RobotML offre également un ensemble d'outils qui visent à faciliter le développement des applications de robotique.
Nous avons vu également que le modèle de domaine RobotML propose des types abtraits, extraits de l'ontologie, appelés *PhysicalData* afin d'abstraire les données des capteurs (voir IV.23).
Cependant, ces types n'ont pas de sémantique d'utilisation précise car ce sont des abstractions a priori qui n'ont pas été définies à partir d'exemples concrets. Il faudrait établir un certain nombre d'itérations à partir d'exemples concrets afin de

convenir d'un ensemble d'abstractions ayant une sémantique bien définie et pouvoir les intégrer éventuellement à l'ontologie. Par conséquent, les méta-classes de *PhysicalData* n'ont pas été étendues par le profil UML et ne sont donc pas visibles pour le moment pour les utilisateurs de RobotML.
Nous nous proposons alors dans le chapitre suivant de présenter une démarche, qui pourrait compléter l'ontologie, avec des abstractions définies a posteriori (c.-à-d. après une étude détaillée d'un exemple de tâche de robotique).

Approche Top-down pour la gestion de la variabilité dans les algorithmes de robotique

1 Introduction

Les roboticiens sont confrontés à plusieurs difficultés pour le développement de leurs applications. La principale difficulté concerne la maîtrise des détails du matériel du robot. En effet, l'implantation d'une tâche de robotique requiert des informations sur les moyens de locomotion du robot, sa morphologie et ses capteurs. Implanter une tâche de robotique qui marcherait pour tous les moyens de locomotion et qui tiendrait compte des entrées de tous les capteurs est un processus très complexe [45]. Il faudrait au moins utiliser des abstractions comme celles proposées par les middleware ou celles proposées par l'ontologie de RobotML.
Cependant, les abstractions proposées par les middleware sont de bas niveau. Il est très difficile de les appliquer dans le cas où l'on souhaite développer une application pour un robot de forme inconnue, dont les capteurs sont inconnus et où les moyens de locomotion le sont aussi. En effet, toutes ces informations doivent être connues afin de calculer les données des capteurs et exécuter les mouvements du robot [45]. D'autre part, les abstractions proposées par l'ontologie du domaine représentent des concepts du domaine [69] définis a priori et non à partir d'exemples concrets. Par conséquent, ces abstractions sont insuffisantes et n'ont pas de sémantique opérationnelle précise à savoir qu'on ne sait pas forcément comment les implanter dans un programme.

Une autre difficulté peut être soulevée lorsqu'il s'agit d'une tâche de robotique où plusieurs stratégies visent à atteindre un même objectif en se basant sur les mêmes hypothèses sont définies comme la famille de navigation Bug [38]. Il s'agit alors de variabilité algorithmique.
Il est difficile de comparer les variantes d'une famille ou même de les comprendre car elles sont proposées par différents auteurs dans la littérature et sont souvent implantées sous différentes plateformes. Si on pouvait définir un algorithme générique pour les variantes d'une famille d'algorithmes, il serait plus facile de les comprendre, de les implanter et plus significatif de les comparer entre elles. À notre connaissance, très peu de travaux se sont intéressés à la gestion de ce type de variabilité. Brugali et al. [119] ont proposé une ligne de produits logicielle pour la gestion de la variabilité au sein des tâches de robotique telles que la localisation, la navigation et l'évitement d'obstacles. Cependant, leurs composants sont interdépendants. En d'autres termes, ils sont spécifiques à un type de capteur en particulier : le capteur Laser. Leur implantation se base sur des messages ROS ce qui reste spécifique aux types fournis par ce middleware.

Afin de faciliter le développement des applications de robotique et les rendre résistantes aux différents types de variabilité cités précédemment, il faudrait alors définir des abstractions qui soient :

1. de plus haut niveau que celles proposées par les middleware : qui encapsulent les mesures et les données retournées par les capteurs et les commandes spécifiques des effecteurs
2. plus précises que celles proposées par l'ontologie : elles doivent avoir une sémantique opérationnelle

Ces abstractions masqueraient alors les détails non algorithmiques et pourraient varier indépendamment des algorithmes. Concernant les détails algorithmiques dans le cas où il s'agit d'une famille d'algorithmes, il faudrait également définir des abstractions qui masquent la variabilité au sein d'une même famille.
Dans cette optique, nous proposons une approche top-down qui, à partir de la description d'une tâche de robotique et des hypothèses sur l'environnement et sur les capacités du robot, produit :

– un ensemble d'abstractions non algorithmiques ainsi que des abstractions algorithmiques

– un algorithme générique défini en fonction de ces abstractions

L'intervention d'un expert de robotique est requise pour l'identification de ces abstractions et leur combinaison pour définir un algorithme générique de façon analogue à un planificateur. Les hypothèses définies sur les capacités du robot ne doivent pas être liées à des capteurs ni à des effecteurs spécifiques. Elles doivent plutôt se baser sur les données requises par l'algorithme. L'expert définit donc des abstractions comme étant des requêtes sur l'environnement et des actions de plus haut niveau que les commandes spécifiques aux effecteurs.

Dans ce chapitre, nous présentons notre approche pour l'identification de ces abstractions et d'un algorithme générique défini en fonction de ces dernières. Nous proposerons ensuite une démarche pour implanter les abstractions ainsi que l'algorithme générique.

Nous appliquerons ensuite notre approche sur une famille d'algorithmes de navigation appelée Bug [38].

2 Approche pour la gestion de la variabilité dans les algorithmes de robotique

À partir d'une tâche de robotique, notre approche permet de produire un algorithme générique configurable en fonction d'un ensemble d'abstractions qui masquent les détails du matériel et les détails algorithmiques. Une fois l'algorithme générique défini, nous proposons une démarche pour l'implantation de ces abstractions et de l'algorithme générique de façon à faire varier l'implantation de ces abstractions sans avoir à modifier l'implantation de l'algorithme générique. Nous présentons dans cette section les différentes étapes de notre approche.

2.1 Identification des abstractions

La forme des robots varie d'un robot à l'autre ce qui affecte le fonctionnement des capteurs surtout s'ils sont placés sur celui-ci comme les capteurs à rayons (range sensors). Le nombre, la nature et la position de ces capteurs varient aussi d'un robot à l'autre. Un capteur Laser a un angle de perception, les Infrarouges ont des rayons de perception, etc. Les capteurs tactiles ont un autre mode de fonctionnement mais permettent d'indiquer par exemple si un obstacle est détecté tout comme les cap-

teurs à rayons mais avec un mode de fonctionnement différent. Nous avons vu que certains middleware ont défini des interfaces pour les range sensors par exemple. Leurs abstractions sont liées aux types des capteurs.
Nous avons mentionné précédemment que les abstractions que nous voulons définir sont de plus haut niveau que celles proposées par les middleware et ont une sémantique par rapport à celles proposées par l'ontologie du domaine et donc RobotML. Les abstractions dans ce travail permettent d'encapsuler les détails impertinents dans l'algorithme représentant la tâche à développer.
Nous distinguons deux types d'abstractions : algorithmiques et non algorithmiques, nos définitions sont les suivantes :

Definition 11 (Abstraction non algorithmique)**.** *Une abstraction non algorithmique est soit une requête qui retourne des informations sur l'environnement y compris le robot soit une action de haut niveau que le robot doit effectuer.*

Definition 12 (Abstraction algorithmique)**.** *Une abstraction algorithmique est liée à la tâche à développer. Elle peut encapsuler un ensemble d'instructions permettant de réaliser une sous-tâche ou des détails liés aux variantes d'un algorithme lorsqu'il s'agit d'une famille d'algorithmes.*

Les abstractions non algorithmiques sont des abstractions qu'on peut capitaliser car elles sont indépendantes des détails algorithmiques d'une tâche. Les abstractions algorithmiques, quant à elles, peuvent dépendre des abstractions non algorithmiques mais restent spécifiques à la tâche à développer.

Lorsqu'il s'agit d'abstractions non algorithmiques, on peut distinguer deux cas :

1. L'abstraction est une requête sur l'environnement y compris le robot. Dans ce cas, cette requête indique par exemple si le robot est face à un obstacle, s'il perçoit de la lumière, etc. Les paramètres de configuration de cette requête sont spécifiés au niveau de leur implantation.
2. L'abstraction est une action de haut niveau permettant au robot d'agir dans son environnement. Cette action peut être simple comme l'action *avancer* ou composée d'actions simples comme *suivre un obstacle dans une direction donnée*. Les variables permettant de spécifier la vitesse ou la direction que doit suivre une action sont passées en paramètres à celle-ci.

Concernant les abstraction algorithmiques, elles sont de plus haut niveau que les abstractions non algorithmiques. Elles font appel à des actions et à des requêtes sur l'environnement afin d'effectuer une sous-tâche de la tâche à implanter. De notre point de vue, l'identification des abstractions ne nécessite pas de connaissances approfondies des capteurs ou des effecteurs physiques d'un robot particulier comme la démarche adoptée par les middleware. Cette étape requiert une connaissance de la tâche à développer d'où l'intervention d'un expert du domaine.
Notre approche s'appuie sur la description d'une tâche de robotique ce qui permet de définir des hypothèses sur l'environnement et sur les capacités que doit avoir un robot pour qu'il puisse exécuter cette tâche.
À partir de la description d'une tâche, un expert en robotique peut identifier des fonctionnalités que le robot doit effectuer en se basant par exemple sur une démarche dirigée par les buts. Ces fonctionnalités font appel à des actions, lancent des requêtes sur l'environnement et effectuent des traitements sur les données récupérées afin de se rapprocher de l'état but de la tâche à réaliser.
Les requêtes sur l'environnment définissent par conséquent la catégorie des capteurs à utiliser. Par exemple, dans une tâche de navigation, l'une des fonctionnalités du robot consiste à éviter les obstacles rencontrés. Il faut que le robot soit capable de détecter cet obstacle. On peut donc définir des hypothèses sur les capacités du robot à indiquer la présence d'un obstacle dans son champs de perception. Ces requêtes pourront ensuite être appliquées sur des capteurs de distance ou de contact par exemple.
Concernant les actions de haut niveau, elles sont également liées à la description de la tâche. Si par exemple le robot doit avancer ou tourner dans un environnement terreste, il doit être équipé par des roues. Nous pouvons alors définir des hypothèses sur les moyens de locomotion requis pour l'accomplissement des actions.
Une fois ces abstractions identifiées, l'expert du domaine combine ensuite les abstractions algorithmiques et les abstractions non algorithmiques afin de définir un algorithme générique.

2.2 Identification de l'algorithme générique

Dans cette étape, un expert en robotique organise les abstractions identifiées précédemment. Un algorithme générique est une séquence d'instructions qui utilisent les abstractions non algorithmiques et les abstractions algorithmiques. Cette

combinaison se fait sous forme d'algorithme ou sous forme de machine à états finis représentant la tâche étudiée. Étant donné que les abstractions définissent des hypothèses sur les capacités du robot à percevoir ou à agir dans son environnement, l'algorithme générique reste stable pour une catégorie de robots. De cette façon, on peut faire varier les implantations des abstractions sans avoir à modifier l'algorithme générique.
Pour une tâche de robotique donnée, on ne peut pas définir un algorithme générique qui restera stable pour toutes les catégories des capteurs et des effecteurs car il faut que ces derniers correspondent aux hypothèses définies sur les abstractions.

Pour résumer, notre approche comporte deux étapes principales :

1. L'identification des abstractions
2. La définition de l'algorithme générique

Dans ces deux étapes, un expert de robotique doit intervenir afin de combiner les abstractions ou les valider. Ainsi, les abstractions obtenues seront extraites à partir d'exemples concrets et pourront être réutilisées. Nous verrons dans la section suivante comment ces abstractions doivent être organisées au niveau de l'implantation.

2.3 Organisation de l'implantation

L'algorithme générique défini en fonction des abstractions est complètement indépendant des détails des capteurs, des effecteurs et des middleware. Aucune propriété spécifique à un middleware ne doit être utilisée au sein de l'application qui implante l'algorithme générique (c.-à-d. protocole de communication, types spécifiques, etc.). On voudrait maintenant organiser l'implantation des abstractions non algorithmiques et des abstractions algorithmiques.
Une abstraction a une sémantique opérationnelle bien définie. Elle peut avoir un type de retour, dans ce cas, le type de retour est un type abstrait (c.-à-d. indépendant des types retournés par le matériel et des types fournis par les middleware). Un **type abstrait** désigne un type primitif (booléen, entier, réel, etc.) ou un type de données ayant des propriétés dont les types sont primitifs. Par exemple, le type *Point* est un type abstrait. Il est caractérisé par trois coordonnées x, y et z de types réels.
Comme nous l'avons défini précédemment, les abstractions algorithmiques sont de plus haut niveau que les abstractions non algorithmiques et dépendent généralement de celles-ci. Il est donc essentiel des faire varier les abstractions non algorithmiques

indépendamment des abstractions algorithmiques afin d'éviter les situations où on pourrait avoir une explosion combinatoire.
Dans ce qui suit, nous présentons d'abord l'implantation des abstractions non algorithmiques puis l'implantation des abstractions algorithmiques et de l'algorithme générique.

2.3.1 Implantation des abstractions non algorithmiques

Afin d'implanter les abstractions non algorithmiques, il faudrait tout d'abord rassembler les abstractions qui se basent sur le (ou les) même(s) capteur(s) physique(s) ou qui agissent sur le (ou les) même(s) effecteur(s) physique(s).
Pour ce faire, nous nous proposons d'utiliser des adaptateurs des capteurs qui implantent les abstractions basées sur un ou plusieurs capteurs. L'implantation des abstractions au sein des adaptateurs permettra de convertir les données physiques capturées par les capteurs en données de types abstraits séparant ainsi l'acquisition des données de leurs traitements.
Un adaptateur de capteur peut prendre en entrée différents capteurs pour calculer la donnée requise en sortie, dans ce cas il s'agit d'un adaptateur mixte.
De la même façon, nous utiliserons des adaptateurs pour les effecteurs qui permettent d'implanter les actions du robot. L'implantation des actions consiste à spécifier les commandes que les effecteurs doivent exécuter.
Pour résumer, les adaptateurs permettent d'implanter les abstractions du matériel en convertissant les données physiques en données abstraites ou en convertissant les actions en commandes spécifiques comme le montre la figure V.44.

Notons que les adaptateurs sont les seules parties de l'application qui sont spécifiques aux capteurs et aux effecteurs physiques. Au niveau de l'implantation, il faut spécifier certains paramètres de configuration du matériel utilisé. Les seules informations requises par le code de l'application sont les abstractions. Il ne doit pas y avoir de lien permanent entre un adaptateur et le code de l'application. Ce lien ne doit être spécifié qu'au niveau de la configuration de l'application (c.-à-d. au niveau du déploiement) afin de rendre l'application la plus indépendante possible des détails du matériel et de permettre une flexibilité pour les changements de ce dernier.
La fréquence d'exécution des composants, la nature des données requises (avec ou sans Buffer par exemple), le mécanisme de transport en particulier et plus généra-

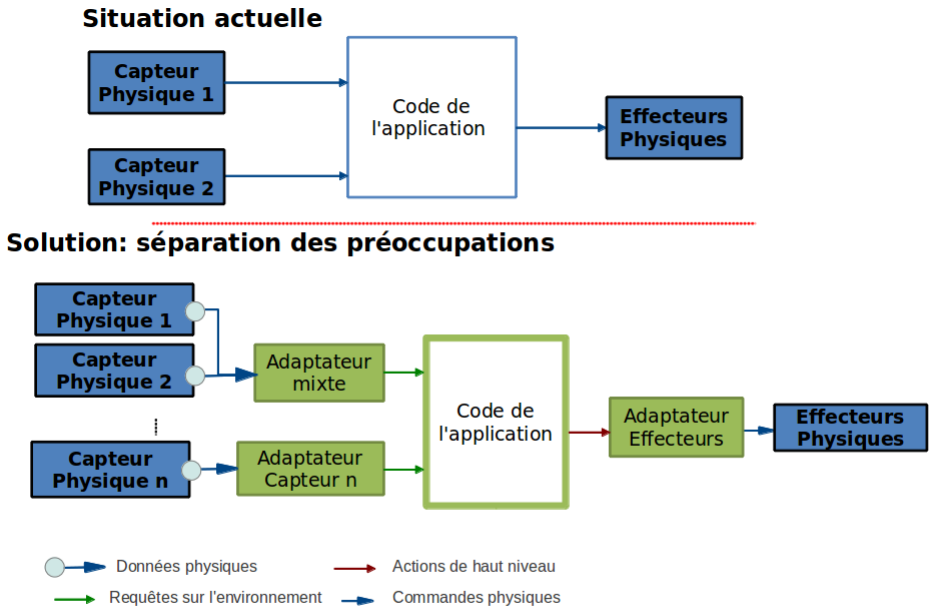

Figure V.44 – Implantation des abstractions non algorithmiques : adaptateurs

lement les paramètres de configuration sont délégués aux middleware. Chacun a un moyen de configuration propre à lui donc cette partie n'est pas spécifiée au niveau de l'implantation des abstractions mais au niveau du déploiement.

Le design pattern Bridge [120] permet de séparer les abstractions de leurs implantations dans différentes hiérarchies de classes. Les abstractions sont définies dans une interface correspondant aux opérations qui doivent être fournies adaptateurs. Les adaptateurs implantent ensuite ces opérations comme le montre la figure V.45. L'avantage de cette approche est que le changement des implantations n'affecte pas le reste du code de l'application. De plus, les changements dans l'implantation des abstractions n'ont aucun impact sur le code de l'application.

2.3.2 Implantation des abstractions algorithmiques et de l'algorithme générique

Prenons maintenant le cas d'une famille d'algorithmes qui a plusieurs variantes. Il faudrait alors définir des abstractions algorithmiques en plus des abstractions non algorithmiques.

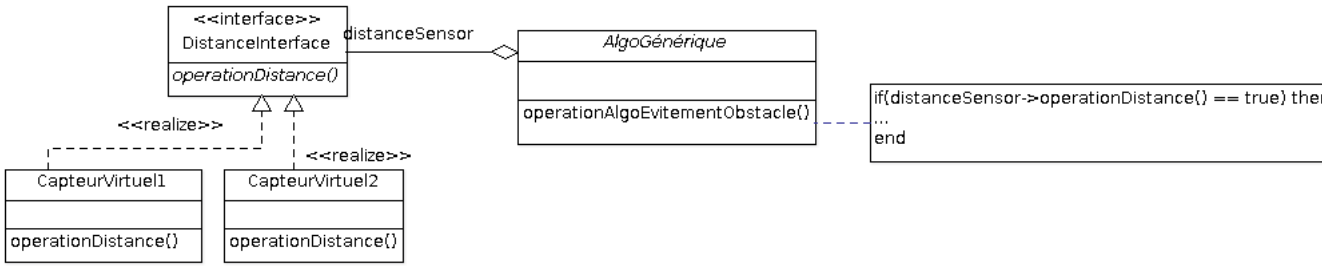

FIGURE V.45 – Application du pattern Bridge

Nous devons séparer l'implantation des abstractions non algorithmiques de l'implantation des abstractions algorithmiques.

Afin d'organiser l'implantation de ces abstractions, nous nous proposons d'utiliser le design pattern Template Method.
Le design pattern Template Method (TM) [120] permet la définition d'un squelette d'algorithme comme une template méthode d'une classe abstraite. Cette méthode n'est autre que l'algorithme générique identifié dans l'étape précédente.
Les points de variations parmi les variantes d'une famille d'algorithmes sont définis en tant que méthodes abstraites, spécialisées dans chaque sous-classe.
Les parties invariantes de l'algorithme sont implantées dans la classe abstraite en tant que méthodes concrètes.
Les instructions qui existent dans certaines variantes et pas dans d'autres sont définies en tant que *hook methods* (avec un corps vide dans la classe abstraite mais qui peut être spécialisé dans les sous-classes) de la classe abstraite.
Par conséquent, Chaque variante de l'algorithme est implantée comme une sous-classe de la classe abstraite et les méthodes abstraites correspondant aux variabilités algorithmiques sont implantées dans chaque sous-classe comme le montre la figure V.46.

Pour résumer, l'organisation de l'implantation combine les deux design pattern Bride et Template Method afin de séparer les abstractions non algorithmiques de celles algorithmiques.

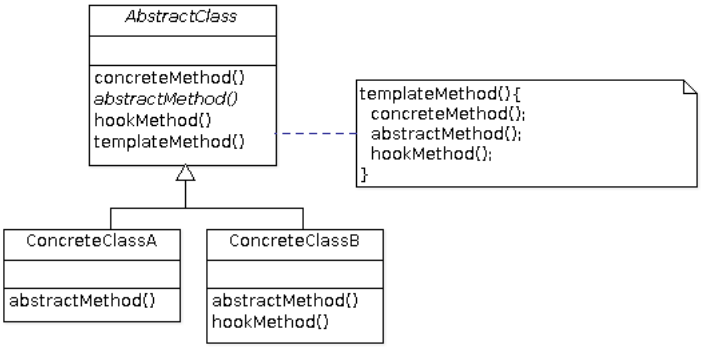

Figure V.46 – Application du pattern Template Method

3 Application sur la famille d'algorithmes Bug

En raison de la variabilité constatée dans les algorithmes de navigation, nous nous sommes intéressés à la famille de navigation Bug [38] où l'environnement du robot est inconnu mais la position du but est connue. Dans cette section, nous commençons d'abord par présenter la famille Bug, nous présenterons par la suite l'application de notre approche sur cette famille.

3.1 La famille d'algorithmes Bug

Bug est une famille d'algorithmes de navigation dans un environnement inconnu. Le robot doit se déplacer de sa position de départ vers son but tout en évitant les obstacles sur son chemin vers celui-ci.
Dans la famille Bug, le robot contourne les obstacles qu'il rencontre suivant différentes stratégies. Certains comportements du robot sont réactifs, par exemple, si un obstacle est détecté, le robot contourne l'obstacle. Si le but est atteint, le robot s'arrête, etc. D'autres comportements s'appuient sur des calculs un peu plus élaborés. Par exemple, le contournement de l'obstacle rencontré consiste à maintenir une distance de sécurité par rapport à celui-ci et à chercher un point potentiel pour quitter cet obstacle.
Dans ce qui suit, nous donnons une description des algorithmes Bug.

1. **Hypothèses sur l'Environnement** : Environnement inconnu 2 dimensions :
   – Un obstacle est un chemin fermé de longueur finie et d'épaisseur non nulle (car l'une des variantes de cette famille utilise l'épaisseur des obstacles dans ses hypothèses).

– Il existe un nombre fini d'obstacles dans l'environnement

2. **Hypothèses sur le robot** :

   (a) **Forme** : Le robot est un point.

   (b) **Perception** :
   – capacité de détecter les obstacles : capteurs tactiles ou à rayons selon les algorithmes.
   – localisation parfaite.

   (c) **Action** :
   – s'orienter vers le but.
   – avancer vers le but.
   – contourner un obstacle.
   – s'arrêter.

   (d) **Notations**
   – **Données en entrée** : q_{start}, q_{goal} ; où q_{start} et q_{goal} sont respectivement la position de départ et du but du robot
   – **Variables** :
   – q_H^i , appelé *hitPoint* dans la littérature, est le point de rencontre d'un obstacle.
   – q_L^i , appelé *leavePoint* dans la littérature, est le point à partir duquel le robot peut quitter l'obstacle rencontré et se dirige de nouveau vers son but.
   – x est la position courante du robot
   – *distance* est la distance par rapport à l'obstacle le plus proche
   – $distMinToGoal$ est la distance euclidienne minimale du robot jusqu'au but.
   – *Step* épaisseur minimale d'un obstacle.
   – $F = \mathrm{r}(\theta)$ est la distance perçue le long d'un rayon r émanant du centre du robot à un angle θ où θ est la direction du but.
   – *direction* est la direction du contournement d'obstacle du robot : dans le sens des aiguilles d'une montre (clockwise) ; contre les sens des aiguilles d'une montre (counter-clockwise).
   – d_{reach} est la distance minimale parmi tous les points perçus par le robot.

– $d_{followed}$ est la distance minimale parmi tous les points autour d'un obstacle et le but.

Nous présentons dans les sections suivantes six algorithmes de la famille Bug : Bug1, Bug2, Alg1, Alg2, DistBug et TangentBug.

3.1.1 Bug1

L'algorithme Bug1 [38, 121] est le premier algorithme proposé dans la famille Bug. La description de Bug1 est donnée dans l'algorithme 1. Bug1 se base sur des capteurs tactiles pour la détection d'obstacles. Lorsque le robot rencontre un obstacle, il contourne l'obstacle à gauche (en le maintenant à sa droite) tout en cherchant le point ayant la distance euclidienne minimale par rapport au but. Il effectue un tour complet autour de l'obstacle pour identifier le point optimal. Une fois ce point identifié, il y retourne et reprend son chemin vers le but. Si le but est inatteignable alors le robot s'arrête et termine sa mission.

Algorithm 1: Bug1 pseudo-code [38, 121]

Sensors : Une méthode de localisation parfaite.
Un capteur de détection d'obstacles
Input : Position de départ (q_{start}), Position du but (q_{goal})
Initialisation: Point $q_L^0 \leftarrow q_{start}$; int $i \leftarrow 1$;

`(1)` A partir de q_L^i avancer vers le but jusqu'à ce que l'une de ces conditions soient satisfaites :
q_{goal} atteint, Stop.
Un obstacle est rencontré, $i \leftarrow i + 1$, $q_H^i \leftarrow x$, aller à l'étape `(2)`.
`(2)` Contourner l'obstacle à gauche tout en cherchant un point y autour de l'obstacle tel que : $d_{euclidian}$ (y, q_{goal}) est minimale.
Si y trouvé, $q_L^i \leftarrow y$, enregistrer le chemin de q_H^i vers q_L^i
S'arrêter lorsque q_H^i est rencontré de nouveau.
Suivre le chemin le plus court pour retourner à q_L^i
Si but inatteignable, retourner échec
Sinon retourner à l'étape `(1)`.

3.1.2 Bug2

La description de Bug2 est donnée dans l'algorithme 2. Bug2 a été défini initialement en utilisant des capteurs tactiles. Dans Bug2 [121], lorsque le robot rencontre un obstacle, il le contourne à gauche et ne le quitte que s'il trouve un point sur une

ligne imaginaire appelée *M-line* passant par son point de départ et le but.
Lorsque ce point est trouvé, le robot vérifie que ce point est plus proche que le point où il a rencontré l'obstacle et si le but est atteignable à partir de celui-ci. Si ces conditions sont satisfaites, le robot s'oriente vers son but et continue son chemin vers celui-ci sinon si aucun point correspondant à ces conditions n'est identifié, l'algorithme indique que le but est inatteignable, le robot s'arrête et termine sa mission.

Algorithm 2: Bug2 pseudo-code [38, 121]

Sensors : Une méthode de localisation parfaite.
Un capteur de détection d'obstacles
Input : Position de départ (q_{start}), Position du but (q_{goal})
Initialisation: Point $q_L^0 \leftarrow q_{start}$; int $i \leftarrow 1$;

`(1)` A partir de q_L^i avancer vers le but en suivant la ligne (q_{start},q_{goal}) jusqu'à ce que l'une de ces conditions soient satisfaites :
q_{goal} atteint, Stop.
Un obstacle est rencontré, $i \leftarrow i + 1$, $q_H^i \leftarrow x$, aller à l'étape `(2)`.
`(2)` Contourner l'obstacle à gauche jusqu'à ce que :
`(A)` q_H^i rencontré de nouveau, retourner échec
`(B)` un nouveau point y est trouvé sur la ligne (q_{start}, q_{goal}) tel que y est plus proche du but que q_H^i
Si but atteignable à partir de y, $q_L^i \leftarrow x$, retourner à l'étape `(1)`
Sinon $d(q_H^i, q_{goal}) \leftarrow d(x, q_{goal})$, continuer à contourner l'obstacle.

3.1.3 Alg1

Alg1 [122, 123] est une extension de Bug2. Sa description est donnée dans l'algorithme 3. Comme Bug1 et Bug2, Alg1 a été défini dans la littérature avec des capteurs tactiles.
À la différence de Bug2, Alg1 enregistre tous les hit points et les leave points qu'il a rencontré sur la *M-line*. L'identification d'un leave point dans Alg1 est similaire à celle dans Bug2 (c.-à-d. un leave point est un point sur la M-line plus proche que tous les points déjà visités).
Dans Alg1, lorsque le robot rencontre un ancien point visité, il retourne au dernier hit point. Une fois le dernier hit point atteint, il contourne l'obstacle dans la direction contraire à sa direction initiale (c.-à-d. contournement à droite) toujours à la recherche d'un leave point potentiel.

Algorithm 3: Alg1 pseudo-code [122]

Sensors : Une méthode de localisation parfaite.
Un capteur de détection d'obstacles
Input : Position de départ (q_{start}), Position du but (q_{goal})
Initialisation: Point $q_L^0 \leftarrow q_{start}$; int $i \leftarrow 1$;

`(1)` A partir de q_L^i avancer vers le but en suivant la ligne (q_{start},q_{goal}) jusqu'à ce que l'une de ces conditions soient satisfaites :
q_{goal} atteint, Stop.
Un obstacle est rencontré, $i \leftarrow i + 1$, $q_H^i \leftarrow x$, aller à l'étape `(2)`.
`(2)` Contourner l'obstacle à gauche jusqu'à ce que :
`(A)` q_H^i rencontré de nouveau, retourner échec
`(B)` un nouveau point y est trouvé sur la ligne (q_{start}, q_{goal}) tel que y est plus proche du but que q_H^i et le but est atteignable à partir de y, $q_L^i \leftarrow x$, retourner à l'étape `(1)`.
`(C)` q_H^j ou q_L^j rencontré tel que ($j < i$), retourner à q_H^i et à partir de q_H^i contourner l'obstacle à gauche. Cette règle ne peut pas être répétée avant qu'un nouveau q_L^i soit défini.

3.1.4 Alg2

Alg2 [124] est une extension de Alg1. Sa description est donnée dans l'algorithme 4. Alg2 se base aussi sur des capteurs tactiles. Le robot ne se base plus sur la condition de la *M-line* pour quitter l'obstacle rencontré mais sur une nouvelle condition qui consiste à trouver le premier point ayant la distance minimale par rapport au but et à partir duquel le but est atteignable. Le robot peut quitter l'obstacle sur ce point et continue son chemin vers le but.

3.1.5 DistBug

Dans DitBug [125], le robot utilise une nouvelle donnée introduite par l'utilisateur ainsi que des capteurs de distance. Cette donnée est l'épaisseur minimale du mur d'un obstacle appelée *STEP*. Le robot utilise ses capteurs de distance et l'épaisseur minimale des obstacles présents dans l'environnement pour trouver un point qui est plus proche du but, à une valeur *STEP* près, que tous les points déjà visités. La description de DistBug est présentée dans l'algorithme 5.

Algorithm 4: Alg2 pseudo-code [124]

Sensors : Une méthode de localisation parfaite.
Un capteur de détection d'obstacles
Input : Position de départ (q_{start}), Position du but (q_{goal})
Initialisation: Point $q_L^0 \leftarrow q_{start}$; int $i \leftarrow 1$; double $distMinToGoal \leftarrow d_{euclidian}\ (q_{start},\ q_{goal})$

`(1)` A partir de q_L^i avancer vers le but en suivant la ligne (q_{start},q_{goal}) en mettant à jour $distMinToGoal$ si $d_{euclidian}\ (x,\ q_{goal}) < distMinToGoal$, jusqu'à ce que l'une de ces conditions soient satisfaites :
q_{goal} atteint, Stop.
Un obstacle est rencontré, $i \leftarrow i + 1$, $q_H^i \leftarrow x$, aller à l'étape `(2)`.
`(2)` Contourner l'obstacle à gauche, vérifier si $d_{euclidian}\ (x,\ q_{goal}) < distMinToGoal$, si oui alors $distMinToGoal \leftarrow d_{euclidian}\ (x,\ q_{goal})$ jusqu'à ce que :
`(A)` q_H^i rencontré de nouveau, retourner échec
`(B)` un nouveau point y est trouvé tel que $d_{euclidian}\ (y,\ q_{goal}) < distMinToGoal$ et le but est atteignable à partir de y, $q_L^i \leftarrow y$, retourner à l'étape `(1)`.
`(C)` q_H^j ou q_L^j rencontré tel que $(j < i)$, retourner à q_H^i et à partir de q_H^i contourner l'obstacle à gauche. Cette règle ne peut pas être répétée avant qu'un nouveau q_L^i soit défini.

Algorithm 5: DistBug pseudo-code [125]

Sensors : Une méthode de localisation parfaite.
Un capteur de détection d'obstacles
Input : Position de départ (q_{start}), Position du but (q_{goal})
Initialisation: Point $q_L^0 \leftarrow q_{start}$; int $i \leftarrow 1$; double $distMinToGoal \leftarrow d_{euclidian}\ (q_{start},\ q_{goal})$; double Step $\leftarrow$ userData

`(1)` A partir de q_L^i avancer vers le but en mettant à jour $distMinToGoal$ si $d_{euclidian}\ (x,\ q_{goal}) < distMinToGoal$, jusqu'à ce que l'une de ces conditions soient satisfaites :
q_{goal} atteint, Stop.
Un obstacle est rencontré, $i \leftarrow i + 1$, $q_H^i \leftarrow x$, aller à l'étape `(2)`.
`(2)` Contourner l'obstacle à gauche, vérifier si $d_{euclidian}\ (x,\ q_{goal}) < distMinToGoal$,
si oui alors $distMinToGoal \leftarrow d_{euclidian}\ (x,\ q_{goal})$ jusqu'à ce que :
`(A)` q_H^i rencontré de nouveau, retourner échec
`(B)` $d_{euclidian}\ (x,\ q_{goal})$ - F <= 0, c-à-d que le but est visible à partir de x, $q_L^i \leftarrow x$, retourner à l'étape `(1)`.
`(C)` $d_{euclidian}\ (x,\ q_{goal})$ - F <= $distMinToGoal$- Step alors $q_L^i \leftarrow x$, retourner à l'étape `(1)`.

3.1.6 TangentBug

L'algorithme Tangent Bug [126] (voir algorithme 6) requiert l'utilisation de capteurs de distance. À partir de n'importe quelle position, le robot doit récupérer sa distance par rapport à l'obstacle le plus proche le long d'un rayon r émanant du centre du robot à un angle θ [] tel que :

$$distance = \varphi(x, \theta) = \min_{\alpha \in [O, \infty)} d(x,\ x\ + \alpha \begin{bmatrix} \cos\theta \\ \sin\theta \end{bmatrix}) \tag{V.1}$$

$$tel\ que\ x\ + \alpha \begin{bmatrix} \cos\theta \\ \sin\theta \end{bmatrix} \in \cup_i Boundary(O_i) \tag{V.2}$$

Cette distance permet au robot de calculer les points discontinuités autour d'un obstacle. Les points de discontinuité sont des points situés sur l'obstacle détectés par les rayons des capteurs du robot et dont la valeur de la distance indique u'un intervalle de valeurs est fini et qu'un autre intervalle a commencé.
Pour chaque point de discontinuité, l'algorithme calcule la distance heuristique qui est égale à la distance du robot vers un point de discontinuité et de ce point de discontinuité vers le but :
$d_heuristic = d(x, P_i) + d(P_i, q_{goal})$; où x est la position courante du robot comme défini plus haut et P_i est un point de discontinuité.
À chaque itération, le robot cherche le point ayant la distance heuristique minimale parmi tous les points de discontinuité et se dirige vers ce point.
À partir de ce point, le robot commence à effectuer le contournement d'obstacle en gardant la même direction que celle qu'il a suivi pour aller à ce point. Il calcule maintenant les deux distances d_{reach} et $d_{followed}$ et s'arrête lorsque $d_{reach} < d_{followed}$ pour se diriger vers son but de nouveau.

3.2 Identification des abstractions de la famille Bug

Les algorithmes Bug sont très similaires fondamentalement mais diffèrent dans certains points. Il existe deux comportements principaux dans les algos de Bug : éviter les obstacles (obstacle avoidance) et se diriger vers le but (motion to goal).

1. Le comportement Motion-to-Goal. Le robot **s'oriente vers son but** et **avance en suivant la direction qui le mène vers celui-ci**. Selon les algorithmes,

Algorithm 6: TangentBug pseudo-code [126]

Sensors : Une méthode de localisation parfaite.
Un capteur de détection d'obstacles
Input : Position de départ (q_{start}), Position du but (q_{goal})
Initialisation: Point $q_L^0 \leftarrow q_{start}$; int $i \leftarrow 1$; double $distMinToGoal \leftarrow d_{euclidian}\ (q_{start}, q_{goal})$

`(1)` A partir de q_L^i avancer vers le but, jusqu'à ce que l'une de ces conditions soient satisfaites :
q_{goal} atteint, Stop.
Un obstacle est rencontré, $i \leftarrow i + 1$, $q_H^i \leftarrow x$, aller à l'étape `(2)`.
`(2)` Calculer points de discontinuités autour de l'obstacle rencontré.
Parmi les points de discontinuité, chercher le point ayant la distance heuristique minimale et la direction vers ce dernier.
Le robot se déplace jusqu'à ce point jusqu'à ce que la distance heuristique ne diminue plus.
Contourner l'obstacle dans la même direction, jusqu'à ce que :
`(A)` q_H^i rencontré de nouveau, retourner échec
`(B)` $d_{reach} < d_{followed}$, $q_L^i \leftarrow y$, retourner à l'étape `(1)`.

Motion-To-Goal peut être trivial ou avec un enregistrement de données qui seront utiles pour le reste de l'algorithme comme le cas de l'algorithme DistBug présenté précédemment par exemple. En effet, dans DistBug le robot doit avancer vers son but en mettant à jour sa distance euclidienne minimale par rapport au but. Cette variable sera utilisée dans la suite de l'algorithme lorsque le robot rencontrera un obstacle pour identifier le meilleur point qui lui permettra de quitter cet obstacle. Nous retrouvons le comportement trivial de Motion-To-Goal dans tous les algorithmes de Bug où il faut que le robot avance simplement vers son but. Le robot **s'arrête**, quitte ce mode et bascule vers le mode Obstacle-Avoidance dès qu'un obstacle est détecté.

2. Le mode Obstacle-Avoidance : Contrairement à Motion-To-Goal, Obstacle-Avoidance est différent d'un algorithme à l'autre mais se base sur les mêmes étapes pour l'identification d'un leave point. Dans tous les algorithmes de Bug, lorsqu'un obstacle est détecté, le robot enregistre sa **position courante** et calcule dans certains cas des données supplémentaires qu'il va utiliser tout au long de l'algorithme. En effet, étant donné que l'environnement inconnu, la position du robot permet de savoir s'il a atteint la position de son but en calculant la distance euclidienne par rapport à celui-ci. D'autre part, sa po-

sition est utile lorsqu'il s'agit de retenir la position d'un point par lequel il est passé. Après l'enregistrement des données, le robot **contourne l'obstacle** jusqu'à ce que la condition d'évitement d'obstacle définie par l'algorithme soit satisfaite (c.-à-d. leave point identifié) ou lorsque l'algorithme indique que le but est inatteignable. Si la condition de l'algorithme est satisfaite, le robot **se dirige vers le leave point identifié**, et reprend son chemin vers le but en rebasculant vers le comportement Motion-To-Goal.

Le robot change de comportement si un **obstacle est détecté** ou si un point potentiel (c.-à-d. leave Point) qui permet de quitter l'obstacle a été **identifié**. L'identification de ce point potentiel représente le point de variabilité principal des algorithmes de Bug. La description informelle de Bug cache la variabilité algorithmique et celle du matériel. La variabilité du matériel a un impact sur la détection d'obstacles et sur la localisation en raison de leurs dépendances des capteurs du robot utilisé. Quant à la variabilité algorithmique, elle est liée à l'identification d'un leave point pour quitter l'obstacle rencontré.
En se basant sur une démarche dirigée par les buts et sur une étude détaillée des algorithmes de Bug, on a identifié les abstractions suivantes :

- Motion-To-Goal : Le robot doit pouvoir se positionner face à son but et être capable d'avancer vers celui-ci : faceGoal(Point q_{goal}), goAhead(double speed)
- Le robot s'arrête : halt()
- La détection d'obstacle sur le chemin du robot : bool obstacleInFrontOfTheRobot() interroge sur les capteurs sur la présence d'un obstacle en face du robot.
- La position courante : Point getPosition()
- Le contournement d'obstacles consiste à suivre une direction (clockwise ou counter-clockwise) en réduisant l'écart entre la distance du robot par rapport au mur suivi et la distance de sécurité comme le montre la figure V.47. Cette abstraction est appelée wallFollowing(bool direction).
  - double getSafeDistance() indique la distance minimale que le robot doit maintenir par rapport à l'obstacle rencontré.
  - obstacleOnTheRight(), bool obstacleOnTheLeft() permettent de situer l'obstacle le plus proche détecté à gauche ou à droite du robot. Imaginons le cas où les capteurs situés à gauche détectent un obstacle et les capteurs droits

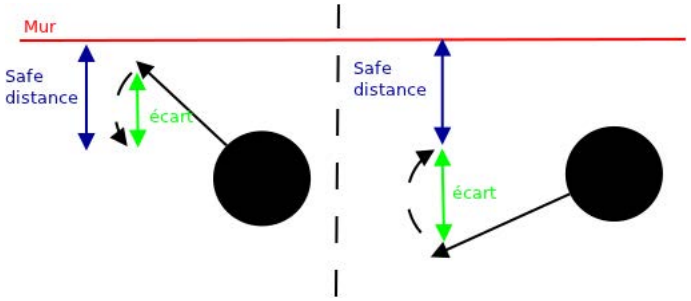

Figure V.47 – Le contournement d'obstacle : (1) le robot à droite doit tourner à droite pour se rapprocher de l'obstacle et réduire l'écart entre la distance de sécurité et sa distance actuelle par rapport à l'obstacle. (2) le robot à gauche doit tourner à gauche pour s'éloigner de l'obstacle et réduire l'écart entre la distance de sécurité et sa distance actuelle par rapport à l'obstacle

détectent un obstacle, il est important de savoir de quel coté se trouve l'obstacle le plus proche. Ces abstractions ne sont pas spécifiques aux capteurs à rayons. Vu qu'elles se présentent sous forme de requêtes qui ne sont pas spécifiques à un type particulier elle peut aussi bien être utilisée pour des capteurs tactiles que pour des capteurs à rayons. Par exemple dans le cas des capteurs à rayons, elles désignent les cadrans à droite et à gauche du robot sur la figure V.48. Ces cadrans peuvent varier selon les champs de vision des capteurs et leurs portées. Cette figure représente le cas où le robot est un point.

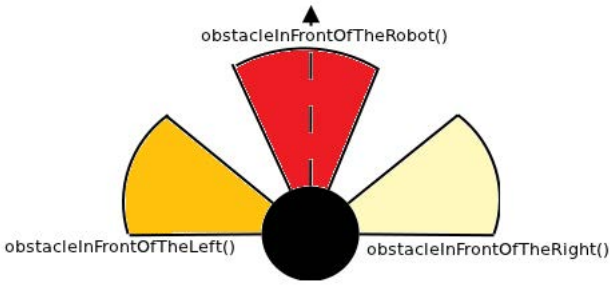

Figure V.48 – La détection d'obstacle devant, à gauche et à droite du robot avec des capteurs de distance

– double getRightDistance(), double getLeftDistance() : afin de maintenir une distance de sécurité par rapport au mur qui représente l'obstacle, nous devons savoir quelle est la distance par rapport à l'obstacle le

plus proche à gauche ou à droite (selon la direction suivie) comme le montre la figure V.49. La distance retournée est la moyenne des distances détectées par les rayons du capteur gauche (ou droit) du robot par rapport à l'obstacle le plus proche.

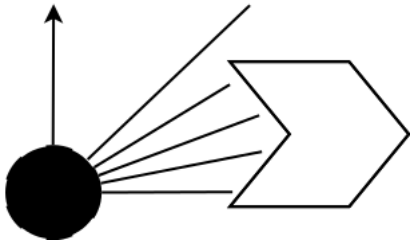

FIGURE V.49 – Exemple de getDistanceRight : la distance retournée est la moyenne des distances détectées parmi tous les rayons émanant du capteur droit du robot

– Trouver un leave Point : IDENTIFYLEAVEPOINT(BOOL DIRECTION, POINT ROBOTPOSITION, POINT GOALPOSITION). La recherche d'un leave point consiste à contourner l'obstacle tout en vérifiant les conditions définies par l'algorithme pour l'identification de celui-ci. La stratégie d'évitement d'obstacle peut se baser sur une décision locale (choisir le premier point qui répond aux conditions de l'algorithme) ou une décision globale (choisir le point parmi tous les points visités qui répond au mieux aux conditions de l'algorithme). Par conséquent, nous avons identifié une abstraction qui sera appliquée à chaque point visité autour de l'obstacle et qui varie d'un algorithme à un autre : FINDLEAVEPOINT(POINT ROBOTPOSITION, POINT HITPOINT, POINT GOALPOSITION). Les données enregistrées varient aussi d'un algorithme à l'autre, dans certains algorithmes on n'enregistre que le point de rencontre de l'obstacle. Dans d'autres, on enregistre les points perçus par le robot ayant une distance heuristique minimale. C'est pour cette raison qu'on a identifié l'abstraction COMPUTEDATA(POINT ROBOTPOS) Le code de IDENTIFYLEAVEPOINT est donné dans l'algorithme 7.

Algorithm 7: Identify leavePoint method

```
identifyLeavePoint(Bool direction, Point robotPos, Point goalPos){
computeData(robotPos) ;
wallFollowing(direction) ;
findLeavePoint(robotPos, getLastHitPoint()) ;
}
```

- Leave point identifié : BOOL ISLEAVEPOINTFOUND() indique si un leave point a été identifié.
- Recherche Complète BOOL RESEARCHCOMPLETE(POINT ROBOTPOSITION, POINT HITPOINT, POINT GOALPOSITION) : Dans le cas où la stratégie appliquée se base sur une décision locale, la recherche est terminée se traduit par l'identification d'un leave point qui répond aux conditions de l'algorithme. En d'autres termes, research complete est équivalent à LEAVEPOINTIDENTIFIED dans le cas d'une décision locale. Lorsqu'il s'agit d'une décision globale, la recherche n'est terminée que lorsque le robot a effectué un tour complet autour de l'obstacle afin d'être sûr qu'il a identifié le meilleur point qui répond aux conditions de l'algorithme.
- Aller au leave point GOTOPOINT(POINT LEAVEPOINT) : Une fois le leave point identifié, le robot se dirige vers celui-ci. Ceci constitue un point de variation parmi les algorithmes car le robot peut par exemple suivre le chemin le plus court pour aller vers ce point ou appliquer d'autres stratégies.
- but inatteignable : Cette condition est vérifiée si aucun leave point n'a été identifié après que le robot ait effectué un tour complet autour de l'obstacle rencontré (NOT ISLEAVEPOINTFOUND() and COMPLETECYCLEAROUNDOBSTACLE(POINT ROBOTPOSITION,POINT HITPOINT)).
- but atteint : Selon le but de l'algorithme, nous définission une marge d'erreur qui indique si le robot doit atteindre son but ou s'il doit s'arrêter peu avant de l'atteindre : BOOL GOALREACHED(POINT ROBOTPOSITION, POINT GOALPOSITION, DOUBLE ERR).

Ces abstractions combinent les abstractions algorithmiques et les abstractions non algorithmiques. Nous pouvons alors les classifier comme suit :

1. Abstractions non algorithmiques : les paramètres de configuration des capteurs sont définis au niveau du déploiement et les caractéristiques spécifiques comme par exemple le nombre de rayons des capteurs, leurs portée, etc. sont spécifiées au niveau des adaptateurs. Etant donné que le robot est considéré comme un point dans les hypothèses de la famille Bug, nous n'allons pas prendre en compte la position des capteurs sur le robot mais simplement de leur emplacement (devant, derrière, à gauche ou à droite) en supposant qu'ils sont placés au même niveau que la plateforme du robot. Les abstractions non algorithmiques sont classées dans 2 sous-catégories :
   - Abstractions non algorithmiques en tant que requêtes sur l'environnement

nécessitant des capteurs à rayons :
- double getSafeDistance()
- double getLeftDistance()
- double getRightDistance()

Nous avons vu que ces abstractions servent au contournement d'obstacle. Elles permettent de retourner la distance à gauche ou à droite du robot par rapport à l'obstacle le plus proche. C'est pour cette raison que nous avons besoin de capteurs à rayons pour pouvoir les implanter et faire des regroupements des rayons des capteurs.

- Abstractions non algorithmiques en tant que requêtes générales :
  - bool obstacleInFrontOfTheRobot()
  - bool obstacleOnTheLeft()
  - bool obstacleOnTheRight()

  Ces abstractions interrogent les capteurs afin de savoir s'il y a un obstacle devant, à droite ou à gauche du robot. Concrètement ces abstractions pourrait être utilisées pour des capteurs à rayons aussi bien que pour des capteurs tactiles ou même pour d'autres types de capteurs tant qu'ils sont capables de founir ces informations. Il n'y a donc aucune contrainte sur ces abstractions qui requiert un type de capteurs particulier.
- Abstractions non algorithmiques en tant qu'actions de haut niveau :
  - halt()
  - faceGoal(Point goalPosition)
  - goAhead(double speed)
  - turnRight(double translation, double angle)
  - turnLeft(double translation, double angle)
  - goToPoint(Point leavePoint)
  - motionToGoal()

2. Abstractions algorithmiques :
   - righHand()
   - leftHand()
   - wallFollowing(bool direction)
   - identifyLeavePoint(bool direction, Point robotPosition, Point goalPosition)
   - findLeavePoint(Point robotPosition, Point hitPoint, Point goalPosition)
   - bool isLeavePointFound()
   - bool researchComplete(Point robotPosition, Point hitPoint, Point goalPosi-

tion)
- bool completeCycleAroundObstacle(Point robotPosition,Point hitPoint)
- goalReached(Point robotPosition, Point goalPosition, double err)

3.3 Identification de l'algorithme générique de la famille Bug

L'algorithme générique est une combinaison des abstractions identifiées dans la section précédente comme une séquence d'instructions. Notre algorithme générique est presenté dans l'algorithme 8.

Algorithm 8: Algorithme générique de la famille Bug

Sensors : Une méthode de localisation parfaite.
Un capteur de détection d'obstacles
input : Position of Start (q_{start}), Position of Target (q_{goal})
Initialisation: robotPos ← `getPosition()` ; direction ← `getDirection()` ;

```
if goalReached(robotPos) then
    EXIT_SUCCESS;
end
else if obstacleInFrontOfTheRobot() == true then
    identifyLeavePoint (direction, robotPos, goalPos);
    if leavePointFound() && researchComplete(robotPos, getHitPoint(),
    q_goal) then
        goToPoint(getLeavePoint());
        faceGoal() ;
    end
    else if completeCycleAroundObstacle(robotPos, getHitPoint())
    && !leavePointFound() then
        EXIT_FAILURE;
    end
end
else
    motionToGoal();
end
```

3.4 Organisation de l'implantation de la famille Bug

Comme nous l'avons mentionné précédemment, le design pattern Template Method permet de gérer le problème de la variabilité en proposant de définir les séquences d'un algorithme dans une template méthode d'une classe abstraite écrite en fonction d'abstractions qui seront implantées dans les sous classes de celle-ci. D'autre part, le design pattern Bridge permet de séparer les abstractions de leur implantation dans différentes hiérarchies de classes.
Chaque variante de l'algorithme est alors implantée dans une sous-classe de la classe abstraite. Quant aux abstractions du matériel, elles sont définies dans des interfaces et implantées par la suite par les adaptateurs correspondant aux capteurs virtuels et aux effecteurs virtuels.
Nous avons besoin d'un adaptateur de capteurs qui va implanter les abstractions suivantes :

- double getSafeDistance()
- bool obstacleInFrontOfTheRobot()
- bool obstacleOnTheLeft()
- bool obstacleOnTheRight()
- double getLeftDistance()
- double getRightDistance()

Cet adaptateur peut prendre en entrée des données d'un ou de plusieurs capteurs physiques capables de founir les informations dont on a besoin. Les paramètres de configuration des capteurs sont définis au sein de ces adaptateurs.
De plus, nous avons besoin d'un adaptateur pour les effecteurs du robot qui va implanter les abstractions suivantes :

- halt()
- faceGoal(Point q_goal)
- goAhead(double speed)
- turnRight(double translation, double angle)
- turnLeft(double translation, double angle)

Prenons maintenant le cas d'un adaptateur Laser ayant un angle de perception de 180° et une portée de 2 mètres. Il faut arriver à extraire les informations de ce capteur afin de les adapter aux besoins des requêtes sur l'environnement. Nous avons alors défini une fonction propre à l'adaptateur appelée getDistanceSensors() qui s'occupe de regrouper les rayons en se basant sur leur nombre et leurs positions (voir

listing V.8).

```
1  vector<double> LaserAdapter::getDistanceSensors(){
2  sensor_msgs::LaserScan msg;
3  vector <double> rays;
4  laser_port.read(msg);
5  int j = 18;
6  int  nbRays = samples / nb_sensors;
7  for(int i=0; i< nb_sensors; i++){
8  rays.push_back(std::accumulate(msg.ranges.begin()+ j - nbRays ,
9                          msg.ranges.begin()+j, 0));
10 j = j + nbRays;
11 }
12 return rays;
13 }
```

Listing V.8 – implantation de getDistanceSensors()

Les autres abstractions implantées dans cet adaptateur utilisent alors ce nouveau regroupement des rayons du capteur afin de renvoyer le résultat de la requête. Par exemple l'implantation de obstacleOnTheLeft() (listing V.9) qui indique si l'obstacle le plus proche est situé à gauche du robot utilise ce regroupement des rayons donné par la fonction getDistanceSensors().

```
1  bool LaserAdapter::obstacleOnTheLeft(){
2          sensor_msgs::LaserScan msg;
3          vector<double> rays = getDistanceSensors();
4          double sensorLeft = rays[4];
5          double sensorLeftFront = rays[3];
6          double sensorRightFront = rays[1];
7          double sensorRight = rays[0];
8
9          return (sensorRight + sensorRightFront >
10                              sensorLeft + sensorLeftFront);
11 }
```

Listing V.9 – implantation de l'abstraction obstacleOnTheLeft()

L'architecture de notre application est présentée dans la figure V.50.

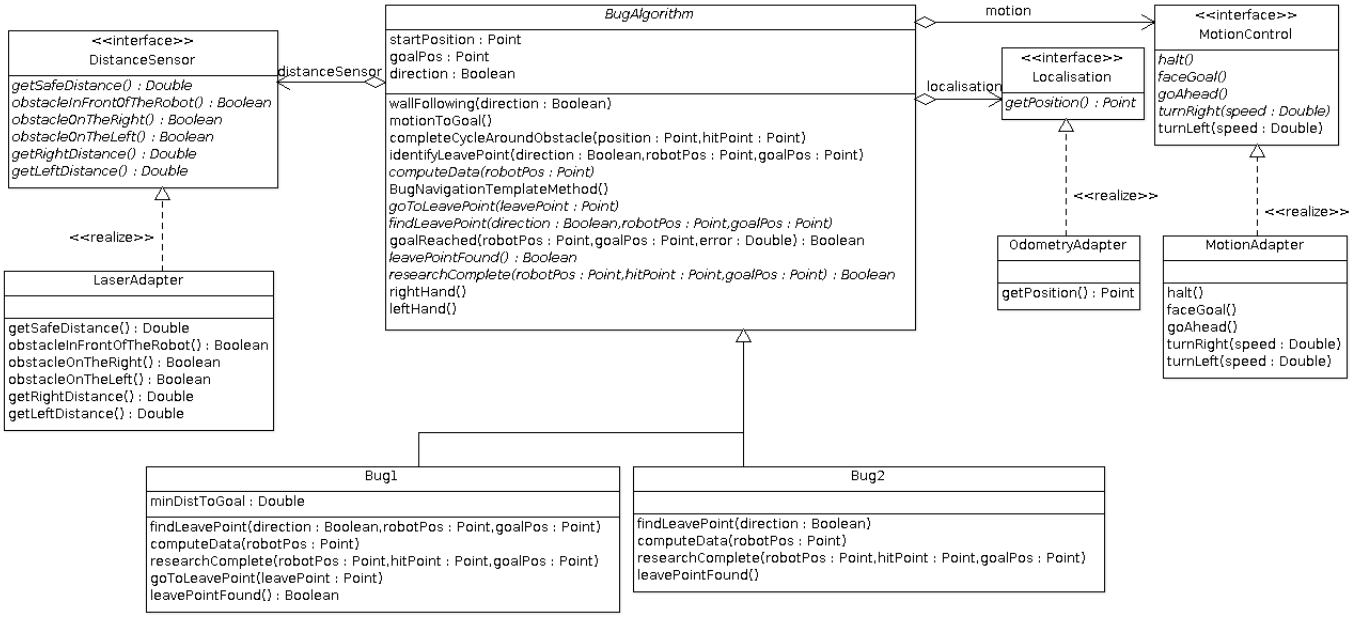

Figure V.50 – Extrait du digramme de classe de la famille Bug

parler de l'implantion de main droite et main gauche : réduire l'écart pour maintenir le mur à droite ou à gauche. Mettre figure mur et orientation robot

Les sources correspondant aux différentes variantes des algorithmes sont disponibles sur https://github.com/SelmaKchir/BugAlgorithms.

3.5 Expérimentations et Validation

Nous avons choisi d'implanter notre application sous OROCOS en utilisant une architecture à base de composants. La classe abstraite `BugAlgorithm` contient la template method représentant l'algorithme générique. Chaque variante de la famille Bug est un composant OROCOS qui implante les méthodes abstraites et qui peut implanter les hook méthodes de la classe abstraite.
Des interfaces correspondant aux abstractions du matériel sont également définies afin de préciser les services qui doivent être fournis par les adaptateurs. Chaque adaptateur est un composant OROCOS lié à un composant matériel (capteur ou actionneur).
Aucun détail spécifique au matériel n'est présenté dans le code de l'application. Les composants sont déployés dans des libraries OROCOS et les liens entre les composants de contrôle et les composants matériels ne sont spécifiés qu'au niveau de

l'exécution à travers un fichier de configuration. Cela garantit une indépendance par rapport aux détails de bas niveau. L'architecture d'une application correspondant à la variante Bug1 de la famille Bug est présentée dans la figure V.51. Plusieurs études

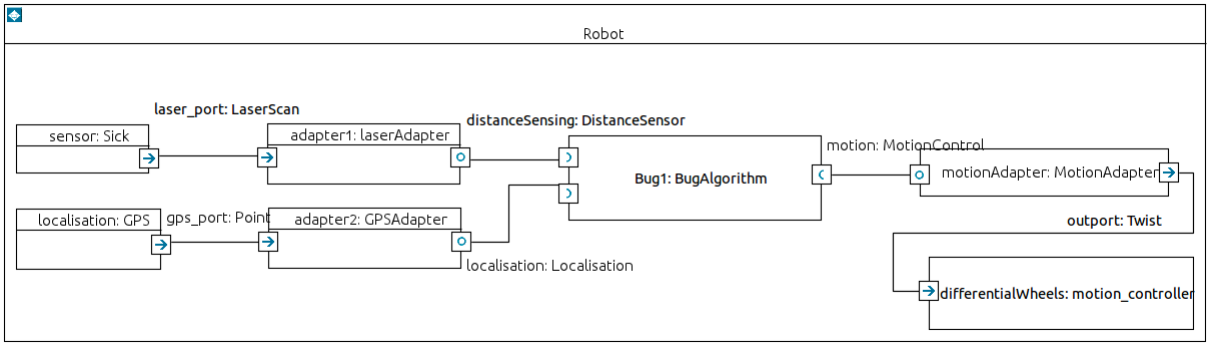

Figure V.51 – Architecture de l'application correspondant à Bug1

comparatives sur les performances des variantes de la famille Bug ont été réalisées dans [127]. Dans cette section, notre objectif n'est pas de comparer ces variantes mais de prouver que chaque variante de Bug peut être implantée avec notre algorithme générique.
Les six variantes présentées précédemment : Bug1 [38], Bug2 [121], Alg1 [122], Alg2 [124], DistBug [125] et TangentBug [126] ont été correctement implantées à partir de notre algorithme générique sous OROCOS.

3.5.1 Environnement de simulation : Stage-ROS

Stage est un environnement de simulation deux dimensions qui fait partie de l'outil de simulation Player/Stage. Il fournit des modèles simples de plusieurs dispositifs. Stage a été intégré à ROS afin de permettre la simulation de programmes développés dans ce middleware ou développés dans d'autres middleware intégrés à ROS. Le noeud stageros simule un monde défini dans un fichier .world, qui contient les détails des capteurs, des robots et des obstacles dans le monde simulé. Stageros étant un noeud de ROS, il est possible de le faire communiquer avec d'autres middleware intégrés à ROS comme OROCOS.
Nous pouvons ainsi simuler notre application implantée avec OROCOS avec Stage en passant pas les messages ROS pour l'intéraction avec les capteurs et les effecteurs physiques du robot simulé.

3.5.2 Configurations des environnements et des capteurs

La simulation a été réalisée au sein de trois environnements.
Le premier environnement présenté dans la figure V.52 nous permet de tester si le robot est capable d'atteindre son but et si l'abstraction GOALISREACHABLE() fonctionne correctement.

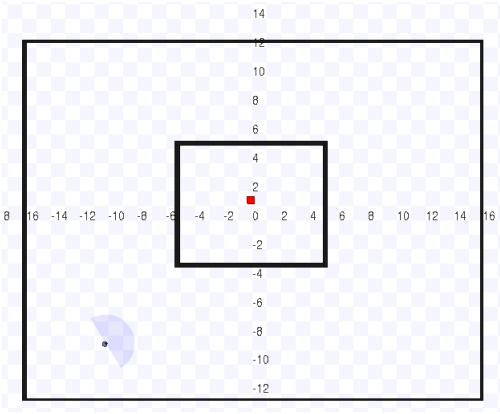

FIGURE V.52 – Environnement1 : but inatteignable

Le second environnement présenté dans la figure V.53 est un environnement de simulation simple avec un seul obstacle. Le but de l'utilisation de cet environnement est de tester simplement le contournement d'obstacle et la capacité du robot d'identifier un point pour le quitter. Nous ne pouvons pas distinguer le comportement du robot dans les algorithmes Bug2 et Alg1 à travers cet environnement car le robot ne revisite pas de points qu'il a déjà visités.

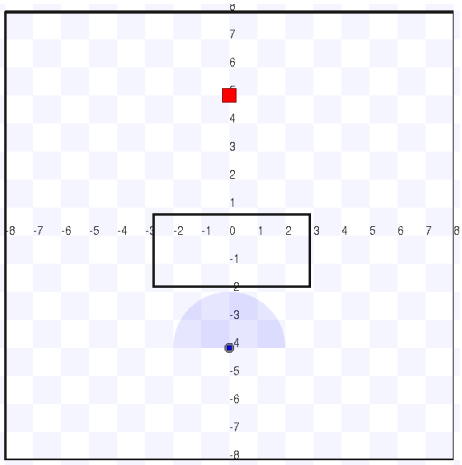

FIGURE V.53 – Environnement2 : Environnement avec un seul obstacle

Nous avons alors défini un troisième environnement V.54 pour pouvoir distinguer les trajectoires d'exécution du robot dans les environnement Bug2 et Alg1 et dans les autres algorithmes.

Concernant les capteurs, nous avons utilisé deux configurations de robots : le pre-

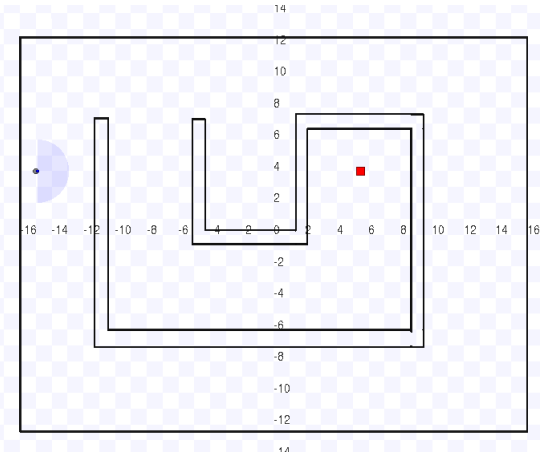

Figure V.54 – Environnement3 : Environnement avec plusieurs obstacles

mier avec un capteur Laser et le deuxième avec trois capteurs infrarouges.

La première configuration a été effectuée en utilisant un capteur Laser ayant un angle de perception de 180 degré et dont les rayons de cet angle ont une portée variant de 0.02 mètres et allant jusqu'à 4 mètres. Pour la localisation, nous avons utilisé un GPS (en supposant que l'environnement de simulation est à l'extérieur).

La deuxième configuration a été réalisée en utilisant trois capteurs infrarouges placés sur l'avant, sur la gauche et sur la droite du robot par rapport à son axe central. Ces capteurs ont un champs de vision (Field Of View) de 26 degré et une portée qui va jusqu'à 2 mètres.

Afin de valider notre démarche nous avons testé nos algorithmes avec deux configurations différentes dans ces trois environnements.

3.5.3 Résultats

Les résultats de simulation montrent une différence de trajectoire entre la première et la deuxième configuration. En effet, la trajectoire suivie par le robot ayant un capteur Laser est différente de celle suivie par le robot ayant les capteurs infrarouges. Ceci est dû au temps d'interprétation des données et aux configurations différentes. Cependant dans tous les cas le robot termine sa mission et ar-

rive à atteindre son but. Concernant les différents algorithmes, ils sont correctement exécutés dans ces trois environnements. Les résultats de simulation sont disponibles sur le lien : https://github.com/SelmaKchir/BugAlgorithms/wiki/Implementing-Bug-Algorithms-variants

4 Discussion

Nous avons démontré que les abstractions identifiées permettent d'être réutilisées indépendamment du robot utilisé grâce à l'utilisation des adaptateurs des capteurs et des effecteurs. Ces abstractions ont une sémantique car elles ont été définies à partir pour une tâche de robotique.
Si nous souhaitons ajouter une nouvelle variante de la famille Bug, il faut implanter les abstractions algorithmiques, les abstractions non algorithmiques quant à elles ne doivent pas forcément être implantées s'il n'y a pas de changement de robot. Dans le cas contraire, il faut ajouter un adaptateur pour les abstractions non algorithmiques. Concernant les requêtes sur l'environnement, nous avons distingué des requêtes générales indépendantes de la classe des capteurs utilisés (range, tactiles, etc.) et des requêtes spécifiques aux capteurs à rayons. Si nous souhaitons supporter les capteurs tactiles par exemple, il faudrait alors changer l'implantation de l'abstraction WALLFOLLOWING vu qu'elle utilisait les abstractions GETLEFTDISTANCE(), GETRIGHTDISTANCE(), GETSAFEDISTANCE() qui s'appuient sur les capteurs à rayons.
Les abstractions algorithmiques résistent aux variations non algorithmiques. En effet, si on prend l'exemple de l'abstraction IDENTIFYLEAVEPOINT (voir algorithme 7), même si on change la façon de contourner l'obstacle et les capteurs qu'on veut utiliser, l'implantation de cette abstraction reste inchangée. Nous avons besoin d'enregistrer plus de données donc on peut les ajouter au niveau de l'implantation de cette variante... L'ajout est alors facile, il suffit de comprendre l'algorithme et de l'intégrer. L'algorithme générique, quant à lui, est stable, il n'y a pas de changement à faire au sein de l'algorithme générique si on veut ajouter une nouvelle variante ou un nouveau matériel.

5 Conclusion

Dans ce chapitre, nous avons proposé une approche qui permet de produire des abstractions algorithmiques, des abstractions non algorithmiques et un algorithme générique défini en fonction de ces dernières. Nous avons présenté également comment implanter ces abstractions de façon à les faire varier dans l'application sans avoir à modifier l'algorithme générique et sans qu'il n'y ait d'explosion combinatoire. En effet, nous avons séparé l'implantation des abstractions non algorithmiques de celles algorithmiques étant donné que ces dernières sont de plus haut niveau et sont spécifiques à la tâche qu'on veut implanter.
Cependant, l'inconvénient d'une telle approche est qu'il faudrait définir autant d'adaptateurs que de capteurs ou d'effecteurs physiques. Afin d'assurer la réutilisabilité de ces abstractions, on pourrait enrichir notre démarche par une librairie d'adaptateurs qui convertissent les données à partir de dispositifs physiques vers les abstractions requises (dans le cas des capteurs) ou inversement (dans le cas des effecteurs). Cette librairie pourrait compléter l'ontologie du domaine proposée dans le cadre du projet PROTEUS en apportant une sémantique aux abstractions définies.

Conclusion générale et perspectives

Le projet PROTEUS a constitué le contexte général de cette thèse dont l'objectif principal est de faciliter le développement des applications de robotique et les rendre résistantes vis-à-vis aux changements des détails de bas niveau. Ce travail de thèse a alors consisté à :

1. Choisir des abstractions indépendantes des plateformes de robotique mobile pour représenter le contrôle et la communication d'une application ;
2. traduire les abstractions de RobotML vers le middleware OROCOS à travers un générateur de code ;
3. proposer une démarche descendante pour rendre les algorithmes résistants aux changements des détails de bas niveau.

Afin de faciliter le développement des applications de robotique, il faudrait permettre aux roboticiens de manipuler des concepts qu'ils ont l'habitude d'utiliser pour le développement de leurs applications. Pour ce faire et afin de traiter le point (1) présenté ci-dessus, nous avons étudié les différents concepts utilisés en robotique. Pour l'aspect contrôle, nous permettons la modélisation de plusieurs types de contrôle (à boucle fermée ou à boucle ouverte) à travers des machines à états finis. Pour l'aspect communication, nous permettons la spécification d'un échange de données synchrone ou asynchrone entre les composants d'une application ainsi que la communication à travers des services.

Les abstractions de RobotML sont proposées à travers un éditeur graphique offrant ainsi une facilité d'utilisation aux roboticiens et même à des simples utilisateurs qui ne sont pas experts en robotique. À partir des modèles représentés avec l'éditeur graphique de RobotML, nous générons le code correspondant vers la plateforme OROCOS. Nous arrivons alors au point (2) ci-dessus. Cette contribution a

consisté à l'élaboration de règles de transformations des abstractions de RobotML vers les concepts d'OROCOS. La génération de code permet d'accélérer le processus de développement d'une application. En effet, le temps passé dans la phase de conception d'une application avec RobotML est largement inférieur au temps passé à développer une application à partir de zéro. Cette contribution a été validée sur un scénario aérien proposé par l'ONERA.

Les abstractions proposées par RobotML sont des abstractions de haut niveau du domaine qui permettent de spécifier les aspects généraux d'une application de robotique : architecture, contrôle et communication. Cependant, elles ne permettent pas de gérer la variabilité liée aux détails de bas niveau des capteurs et des effecteurs d'une part. D'autre part, elles ne permettent pas d'encapsuler les détails algorithmiques d'une application car elles manquent de sémantique opérationnelle. En effet, l'approche adoptée pour la définition des abstractions est une approche qui se base sur les connaissances du domaine pour l'identification des abstractions. Le dernier volet des travaux de cette thèse a été alors de proposer une approche qui permet de définir des abstractions ayant une sémantique opérationnelle.
Nous sommes partis de la description d'une tâche de robotique afin d'identifier ce que nous avons appelés des abstractions non algorithmiques et des abstractions algorithmiques. Les abstractions non algorithmiques concernent essentiellement le matériel et se présentent sous forme de requêtes sur l'environnement ou des actions de haut niveau. Les abstractions algorithmiques sont de plus haut niveau que les abstractions non algorithmiques, elles encapsulent les détails algorithmiques relatifs à une sous-tâche de robotique et utilisent les abstractions non algorithmiques. Un expert de robotique combine ensuite ces abstractions afin de définir un algorithmique générique qui résiste aux changements de bas niveau ainsi qu'aux variations algorithmiques. Notre approche a été validée sur une famille d'algorithmes de navigation appelée Bug [38]. Nous avons fait varier les capteurs utilisés ainsi que les stratégies proposées par six variantes de cette famille et nous avons démontré que l'algorithme générique reste inchangé malgré ces variations.

Une perspective immédiate de nos travaux serait d'enrichir les abstractions de RobotML avec une librairie d'abstractions contenant celles que nous avons identifiées pour une famille de navigation. Ainsi, il devient possible au niveau de la modélisation d'utiliser des abstractions qu'on sait comment implanter. Une deuxième perspective

consiste à appliquer notre approche sur une autre étude de cas afin d'identifier davantage d'abstractions.

L'implantation de notre approche pour la définition d'un algorithme générique en fonction d'abstractions ayant une sémantique opérationnelle se base sur les patrons de conception Template Method et Bridge. Il n'est pas encore possible avec notre approche de définir des contraintes sur les algorithmes comme par exemple les capteurs requis qui peuvent être des capteurs de distance dans certains cas. Une perspective possible serait d'utiliser les lignes de produits algorithmiques dans ce contexte afin de définir des contraintes sur l'association entre les capteurs requis pour des abstractions algorithmiques particulières. Une ébauche de cette perspective a été réalisée dans le cadre de cette thèse mais les résultats sont insuffisants pour être présentés dans ce travail.

Une perspective de recherche à long terme des travaux de cette thèse serait de s'intéresser à un autre sous-domaine de la robotique comme par exemple les robots manipulateurs. À partir de notre expérience avec RobotML, nous pouvons savoir quelles sont les abstractions qui peuvent être réutilisées et celles qu'il faut ajouter afin de faciliter le développement de ce sous-domaine.

Références bibliographiques

[1] Ronald C. Arkin. *Behavior Based Robotics (Intelligent Robotics and Autonomous Agents)*. The MIT Press (1998). vii, 7, 8, 13, 14, 15

[2] Mark Strembeck and Uwe Zdun. *An approach for the systematic development of domain-specific languages*. Softw. Pract. Exper. **39**(15), 1253–1292 October (2009). vii, 35, 41, 44, 45

[3] Herman Bruyninckx, Markus Klotzbücher, Nico Hochgeschwender, Gerhard Kraetzschmar, Luca Gherardi, and Davide Brugali. The brics component model : a model-based development paradigm for complex robotics software systems. In *Proceedings of the 28th Annual ACM Symposium on Applied Computing*, SAC '13, pages 1758–1764, New York, NY, USA (2013). ACM. vii, 2, 48, 50, 58, 60, 84

[4] Soraya Arias, Florine Boudin, Roger Pissard-Gibollet, and Daniel Simon. ORCCAD, robot controller model and its support using Eclipse Modeling tools. In *5th National Conference on Control Architecture of Robots*, Douai, France May (2010). vii, 2, 50, 51, 58, 60

[5] Smartsoft metamodel. http://www.program-transformation.org/pub/GPCE11/ConferenceProgram/slides-gpce11-steck.pdf. vii, 52

[6] Diego Alonso, Cristina Vicente-Chicote, Francisco Ortiz, Juan Pastor, and Barbara Alvarez. *V^3CMM : a 3-View Component Meta-Model for Model-Driven Robotic Software Development*. Journal of Software Engineering for Robotics **1**(1), 3–17 (2010). vii, 53, 54, 58, 60

[7] Saadia Dhouib, Nicolas Du Lac, Jean-Loup Farges, Sebastien Gerard, Miniar Hemaissia-Jeannin, Juan Lahera-Perez, Stephane Millet, Bruno Patin, and Serge Stinckwich. Control architecture

concepts and properties of an ontology devoted to exchanges in mobile robotics. In *6th National Conference on Control Architectures of Robots* (2011). vii, 65, 67, 68

[8] Roland Siegwart and Illah R. Nourbakhsh. *Introduction to Autonomous Mobile Robots*. Bradford Company, Scituate, MA, USA (2004). xi, 10, 11

[9] Henrik I. Christensen and Gregory D. Hager. Sensing and estimation. In Bruno Siciliano and Oussama Khatib, editors, *Springer Handbook of Robotics*, pages 87–107. Springer (2008). xi, 10

[10] Morgan Quigley, Ken Conley, Brian P. Gerkey, Josh Faust, Tully Foote, Jeremy Leibs, Rob Wheeler, and Andrew Y. Ng. Ros : an open-source robot operating system. In *ICRA Workshop on Open Source Software* (2009). 1, 19, 23

[11] H. Utz, S. Sablatnog, S. Enderle, and G. Kraetzschmar. *Miro - middleware for mobile robot applications*. Robotics and Automation, IEEE Transactions on **18**(4), 493–497 (2002). 1, 19, 25

[12] Douglas Blank, Deepak Kumar, Lisa Meeden, and Holly Yanco. *Pyro : A python-based versatile programming environment for teaching robotics*. J. Educ. Resour. Comput. **4**(3) December (2003). 1, 19, 29, 31

[13] Toby H. J. Collett and Bruce A. Macdonald. Player 2.0 : Toward a practical robot programming framework. In *in Proc. of the Australasian Conference on Robotics and Automation (ACRA* (2005). 1, 19, 20

[14] Arie van Deursen, Paul Klint, and Joost Visser. *Domain-specific languages : an annotated bibliography*. SIGPLAN Not. **35**(6), 26–36 June (2000). 1, 34, 35, 39, 40, 45, 46

[15] Jean Bézivin. *On the unification power of models*. Software and Systems Modeling **4**(2), 171–188 (2005). 2, 35

[16] Christian Schlegel, Andreas Steck, and Alex Lotz. *Model-driven software development in robotics : Communication patterns as key for a robotics component model*. Introduction to Modern Robotics (2012). 2, 52, 56, 58, 60

[17] Michael Brady. *Artificial intelligence and robotics*. Artif. Intell. **26**(1) apr (1985). 7

[18] Nils J. Nilsson. A mobius automation : an application of artificial intelligence techniques. In *Proceedings of the 1st international joint conference*

on Artificial intelligence, IJCAI'69, pages 509–520, San Francisco, CA, USA (1969). Morgan Kaufmann Publishers Inc. 8, 12

[19] Allen Newell, J. C. Shaw, and Herbert A. Simon. Report on a general problem-solving program. In *IFIP Congress*, pages 256–264 (1959). 8

[20] Robin R. Murphy. *Introduction to AI Robotics*. MIT Press, Cambridge, MA, USA, 1st edition (2000). 8

[21] A. Elkady and T. Sobh. *Robotics middleware : A comprehensive literature survey and attribute-based bibliography*. Journal of Robotics **2012** (2012). 9, 19, 20, 31

[22] Jacob Fraden. *Handbook of Modern Sensors : Physics, Designs, and Applications (Handbook of Modern Sensors)*. SpringerVerlag (2003). 9, 10, 11

[23] J.P. Laumond, S. Sekhavat, and F. Lamiraux. *Robot motion planning and control* chapter Guidelines in nonholonomic motion planning for mobile robots, pages 1–53. Lectures. Notes in Control and Information Sciences 229. Springer, N.ISBN 3-540-76219-1 (1998). 11

[24] David Kortenkamp and Reid G. Simmons. Robotic systems architectures and programming. In *Springer Handbook of Robotics*, pages 187–206. (2008). 12, 15

[25] Georges Giralt, Raja Chatila, and Marc Vaisset. An integrated navigation and motion control system for autonomous multisensory mobile robots. In IngemarJ. Cox and GordonT. Wilfong, editors, *Autonomous Robot Vehicles*, pages 420–443. Springer New York (1990). 12

[26] R. Chatila and J. Laumond. Position referencing and consistent world modeling for mobile robots. , **2**, pages 138–145 (1985). 12

[27] James S. Albus, Harry G. McCain, and Ronald Lumia. Nasa/nbs standard reference model for telerobot control system architecture (nasrem). (1989). 13

[28] Ronald C. Arkin and Ronald Arkin. Reactive robotic systems, (1995). 13

[29] Maja J. Mataric. *Behavior-based control : Examples from navigation, learning, and group behavior*. Journal of Experimental and Theoretical Artificial Intelligence **9**, 323–336 (1997). 13

[30] Jonathan Connell. A colony architecture for an artificial creature. Technical report, Cambridge, MA, USA (1989). 13, 14

[31] David Zeltzer and Michael B. Johnson. *Motor planning : An architecture for specifying and controlling the behaviour of virtual actors.* The Journal of Visualization and Computer Animation **2**(2), 74–80 (1991). 14

[32] R.A. Brooks. A robot that walks ; emergent behaviors from a carefully evolved network. In *Robotics and Automation, 1989. Proceedings., 1989 IEEE International Conference on*, pages 692–4+2 vol.2 (1989). 14

[33] Rodney A. Brooks. *A robust layered control system for a mobile robot.* IEEE Journal of Robotics and Automation **2**(10) (1986). 14

[34] Julio Rosenblatt. Damn : A distributed architecture for mobile navigation - thesis summary. In *Journal of Experimental and Theoretical Artificial Intelligence*, pages 339–360. AAAI Press (1995). 14

[35] Pattie Maes. The dynamics of action selection. In *Proceedings of the 11th international joint conference on Artificial intelligence - Volume 2*, IJCAI'89, pages 991–997, San Francisco, CA, USA (1989). Morgan Kaufmann Publishers Inc. 14

[36] Ronald C. Arkin. Path planning for a vision-based autonomous robot. Technical report, Amherst, MA, USA (1986). 15

[37] Ronald Craig Arkin. *Towards cosmopolitan robots : intelligent navigation in extended man-made environments.* Thèse de Doctorat, (1987). AAI8805889. 15

[38] Vladimir J. Lumelsky and Alexander A. Stepanov. Effect of uncertainty on continuous path planning for an autonomous vehicle. , **23**, pages 1616 –1621 dec. (1984). 16, 100, 101, 108, 110, 111, 125, 132

[39] David Bakken. *Middleware.* Encyclopedia of Distributed Computing (2001). 19

[40] James Kramer and Matthias Scheutz. *Development environments for autonomous mobile robots : A survey.* Auton. Robots **22**(2), 101–132 feb (2007). 19

[41] N. Mohamed, J. Al-Jaroodi, and I. Jawhar. Middleware for robotics : A survey. In *Robotics, Automation and Mechatronics, 2008 IEEE Conference on*, pages 736–742 (2008). 19, 20

[42] Richard T. Vaughan, Brian P. Gerkey, and Andrew Howard. On device abstractions for portable, reusable robot code. In *Proc. of the IEEE/RSJ Intl. Conf. on Intelligent Robots and Systems (IROS)*, page 2121–2427, Las Vegas, Nevada (2003). 19, 20

[43] ROS : Robot-Operating System. http://www.ros.org/. 19, 23

[44] PYRO : Python Robotics. http://pyrorobotics.com/. 19, 29

[45] William D. Smart. Is a Common Middleware for Robotics Possible ? In *IEEE/RSJ Int. Conf. on Intelligent Robots and Systems (IROS'07) Workshop on Measures and Procedures for the Evaluation of Robot Architectures and Middleware* November (2007). 23, 99

[46] ROS - Topics. http://wiki.ros.org/Topics. 23

[47] RTT : Real-Time Toolkit. http://www.orocos.org/rtt. 23, 24, 85

[48] ROS - Messages. http://wiki.ros.org/Messages. 24

[49] Corba. http://www.corba.org/. 26

[50] N. Sünderhauf R. Baumgartl P. Protzel D. Krüger, I. Lil. Using and extending the miro middleware for autonomous robots. In *Towards Autonomous Robotic Systems (TAROS), Guildford, September 2006.* (2006). 28

[51] Michael Jackson. *Specializing in software engineering.* IEEE Softw. **16**(6), 120–121,119 November (1999). 33

[52] Marjan Mernik, Jan Heering, and Anthony M. Sloane. *When and how to develop domain-specific languages.* ACM Comput. Surv. **37**(4), 316–344 December (2005). 34

[53] Diomidis Spinellis. Notable design patterns for domain-specific languages, (2001). 34

[54] Richard B. Kieburtz, Laura McKinney, Jeffrey M. Bell, James Hook, Alex Kotov, Jeffrey Lewis, Dino P. Oliva, Tim Sheard, Ira Smith, and Lisa Walton. A software engineering experiment in software component generation. In *Proceedings of the 18th international conference on Software engineering*, ICSE '96, pages 542–552, Washington, DC, USA (1996). IEEE Computer Society. 34

[55] Markus Voelter, Sebastian Benz, Christian Dietrich, Birgit Engelmann, Mats Helander, Lennart C. L. Kats, Eelco Visser, and Guido Wachsmuth. *DSL Engineering - Designing, Implementing and Using Domain-Specific Languages.* dslbook.org (2013). 34, 35, 42

[56] Arie van Deursen and Paul Klint. *Little languages : little maintenance.* Journal of Software Maintenance **10**(2), 75–92 March (1998). 35

[57] David A. Ladd and Christopher J. Ramming. Two Application languages in software production. In *VHLLS'94 : Proceedings of the USENIX 1994 Very High Level Languages Symposium Proceedings on USENIX 1994 Very High Level Languages Symposium Proceedings*, page 10, Berkeley, CA, USA (1994). USENIX Association. 35

[58] Francis Neelamkavil. *Computer simulation and modelling*. John Wiley & Sons, Inc., New York, NY, USA (1987). 35

[59] Omg. http://www.omg.org/. 36

[60] Richard C. Gronback. *Eclipse Modeling Project : A Domain-Specific Language (DSL) Toolkit*. Addison-Wesley Professional, 1 edition (2009). 37, 50

[61] Papyrus. http://www.eclipse.org/papyrus/. 37, 81, 82

[62] Uml. http://www.uml.org/. 37

[63] L. Fuentes-Fernández and A. Vallecillo-Moreno. *An Introduction to UML Profiles*. UPGRADE, European Journal for the Informatics Professional **5**(2), 5–13 April (2004). 37, 43

[64] OMG MOF 2 XMI Mapping Specification, Version 2.4.1, August (2011). 38

[65] Epsilon. http://www.eclipse.org/epsilon/. 39

[66] Kermeta. http://www.kermeta.org/overview/model_transformation. 39

[67] Acceleo. http://www.eclipse.org/acceleo/. 39, 85

[68] Tom-emf. http://tom.loria.fr/wiki/index.php5/Documentation:EMF. 39

[69] Paul Oldfield. Domain modelling, (2002). 39, 99

[70] G. Arango. Domain analysis : from art form to engineering discipline. In *Proceedings of the 5th international workshop on Software specification and design*, IWSSD '89, pages 152–159, New York, NY, USA (1989). ACM. 40

[71] Mark A. Simos. Organization domain modeling (odm) : Formalizing the core domain modeling life cycle, (1996). 40

[72] K. C. Kang, S. G. Cohen, J. A. Hess, W. E. Novak, and A. S. Peterson. Feature-oriented domain analysis (foda) feasibility study. Technical report Carnegie-Mellon University Software Engineering Institute November (1990). 41

[73] Nicola Guarino. Formal ontology and information systems. pages 3–15. IOS Press (1998). 41

[74] Saadia Dhouib, Selma Kchir, Serge Stinckwich, Tewfik Ziadi, and Mikal Ziane. Robotml, a domain-specific language to design, simulate and deploy robotic applications. , **7628**, pages 149–160. Springer Berlin Heidelberg (2012). 46

[75] Piotr Trojanek. *Model-driven engineering approach to design and implementation of robot control system.* CoRR **abs/1302.5085** (2013). 47

[76] Xavier Blanc, Jérôme Delatour, and Tewfik Ziadi. Benefits of the MDE approach for the development of embedded and robotic systems. In *Proceedings of the 2nd National Workshop on "Control Architectures of Robots : from models to execution on distributed control architectures" (CAR 2007)* (2007). 47

[77] Brics. http://www.best-of-robotics.org/. 48

[78] Bride. http://www.best-of-robotics.org/bride/. 49

[79] D. Simon, B. Espiau, K. Kapellos, and R. Pissard-Gibollet. *Orccad : software engineering for real-time robotics. a technical insight.* Robotica **15**(1), 111–115 January (1997). 50

[80] D. Simon, R. Pissard-Gibollet, and S. Arias. Orccad, a framework for safe robot control design and implementation. In *1st National Workshop on Control Architectures of Robots : software approaches and issues CAR'06*, Montpellier (2006). 50

[81] Gerard Berry, Georges Gonthier, Ard Berry Georges Gonthier, and Place Sophie Laltte. The esterel synchronous programming language : Design, semantics, implementation, (1992). 50

[82] Christian Schlegel. *Communication Patterns as Key Towards Component-Based Robotics.* International Journal of Advanced Robotic Systems **3**(1), 49–54 March (2006). 51, 79

[83] Smartsoft reference implementation. http://smart-robotics.sourceforge.net/. 51

[84] C. Schlegel, A. Lotz, and A. Steck. Smartsoft : The state management of a component. Technical report Hochschule Ulm January (2011). 52

[85] Cristina Vicente-Chicote, Fernando Losilla, Bárbara Álvarez, Andrés Iborra, and Pedro Sánchez. *Applying mde to the development of flexible and reusable wireless sensor networks.* Int. J. Cooperative Inf. Syst. **16**(3/4), 393–412 (2007). 54

[86] Genom3. http://homepages.laas.fr/mallet/soft/architecture/genom3/. 55, 58, 60

[87] S. Fleury, M. Herrb, and R. Chatila. Genom : a tool for the specification and the implementation of operating modules in a distributed robot architecture. , **2**, pages 842–849 vol.2 (1997). 55

[88] Anthony Mallet, Cédric Pasteur, Matthieu Herrb, Séverin Lemaignan, and François Felix Ingrand. Genom3 : Building middleware-independent robotic components. In *ICRA*, pages 4627–4632. IEEE (2010). 56

[89] Robotic Technology Component (RTC), August (2012). 56, 58, 60

[90] Byoungyoul Song, Seugwoog Jung, Choulsoo Jang, and Sunghoon Kim. An introduction to robot component model for opros (open platform for robotic services). (2008). 56, 57

[91] Choulsoo Jang, Byoungyoul Song, Seungwoog Jung, Sunghoon Kim, Byeongcheol Choi, Hyo-Young Lee, and Cheol-Hoon Lee. *A development of software component framework for robotic services.* Convergence Information Technology, International Conference on **0**, 1–6 (2009). 56

[92] Mi-sook Kim and Hong Seong Park. Open platform for ubiquitous robotic services. In *Proceedings of the 2012 ACM Conference on Ubiquitous Computing*, UbiComp '12, pages 892–893, New York, NY, USA (2012). ACM. 56

[93] N. Ando, T. Suehiro, K. Kitagaki, T. Kotoku, and Woo-Keun Yoon. RT-middleware : distributed component middleware for RT (robot technology). In *2005 IEEE/RSJ International Conference on Intelligent Robots and Systems*, pages 3933–3938. IEEE (2005). 56, 57

[94] Gostai rtc. http://www.gostai.com/products/rtc/. 57

[95] Proteus. http://www.anr-proteus.fr/. 64

[96] Dassault aviation. http://www.dassault-aviation.com/fr/. 64

[97] ECA : Etudes et Constructions Aéronotiques. http://www.eca-robotics.com/. 64

[98] CEA : Commissariat de l'Energie Atomique. http://www.cea.fr/. 64

[99] GOSTAI. http://www.gostai.com/. 64

[100] Intempora. http://www.intempora.com/. 64

[101] Thales. https://www.thalesgroup.com/fr. 64

[102] Lasmea. http://tims.isima.fr/lasmea.php. 64

[103] Thales optronoique sa. https://www.thalesgroup.com/fr/worldwide/defense/notre-offre-forces-terrestres-c4isr/optronique. 64

[104] Greyc. https://www.greyc.fr/. 64

[105] Inria. http://www.inria.fr/. 64

[106] Onera. http://www.onera.fr/. 64

[107] Prisme. http://www.univ-orleans.fr/prisme. 64

[108] Effidence. http://effistore.effidence.com/. 64

[109] Wifibot. http://www.wifibot.com/. 64

[110] Lip6 : Laboratoire d'Informatique de Paris 6. http://www.lip6.fr. 64

[111] Thomas R. Gruber. *Toward principles for the design of ontologies used for knowledge sharing*. Int. J. Hum.-Comput. Stud. **43**(5-6), 907–928 dec (1995). 65

[112] Gaëlle Lortal, Saadia Dhouib, and Sébastien Gérard. Integrating ontological domain knowledge into a robotic DSL. In *Proceedings of the 2010 international conference on Models in software engineering*, MODELS'10, Berlin, Heidelberg (2011). Springer-Verlag. 69

[113] RTMAPS : Real-Time, Multisensor, Advanced, Prototyping Software. http://www.intempora.com/rtmaps4/rtmaps-software/overview.html. 85

[114] URBI : Universal Real-time Behavior Interface. http://www.urbiforge.org/. 85

[115] Arrocam. http://effistore.effidence.com/. 85

[116] Morse. http://www.openrobots.org/wiki/morse/. 85

[117] Cycabtk. http://cycabtk.gforge.inria.fr/. 85

[118] Rtt-lua. http://www.orocos.org/wiki/orocos/toolchain/luacookbook. 87

[119] Davide Brugali, Luca Gherardi, A. Biziak, Andrea Luzzana, and Alexey Zakharov. A Reuse-Oriented Development Process for Component-Based Robotic Systems. , **7628**, pages 361–374. Springer (2012). 100

[120] Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides. *Design Patterns : Elements of reusable object-oriented software*. Addison-Wesley Publishing (1995). 106, 107

[121] V. Lumelsky and A. Stepanov. *Dynamic path planning for a mobile automaton with limited information on the environment*. Automatic Control, IEEE Transactions on **31**(11), 1058 – 1063 nov (1986). 110, 111, 125

[122] H. Noborio, K. Fujimura, and Y. Horiuchi. A comparative study of sensor-based path-planning algorithms in an unknown maze. , **2**, pages 909 –916 vol.2 (2000). 111, 112, 125

[123] A. Sankaranarayanan and M. Vidyasagar. A new path planning algorithm for moving a point object amidst unknown obstacles in a plane. In *Robotics and Automation, 1990. Proceedings., 1990 IEEE International Conference on*, pages 1930 –1936 vol.3 may (1990). 111

[124] A. Sankaranarayanar and M. Vidyasagar. Path planning for moving a point object amidst unknown obstacles in a plane : a new algorithm and a general theory for algorithm development. In *Decision and Control, 1990., Proceedings of the 29th IEEE Conference on*, pages 1111 –1119 vol.2 dec (1990). 112, 113, 125

[125] Ishay Kamon. Sensory based motion planning with global proofs. In *In Proceedings of the IROS95*, pages 435–440 (1995). 112, 113, 125

[126] Ishay Kamon, Elon Rimon, and Ehud Rivlin. *Tangentbug : A range-sensor-based navigation algorithm*. The International Journal of Robotics Research **17**(9), 934–953 September (1998). 114, 115, 125

[127] James Ng and Thomas Bräunl. *Performance comparison of bug navigation algorithms*. J. Intell. Robotics Syst. **50**(1), 73–84 September (2007). 125

Résumé L'un des challenges des roboticiens consiste à gérer un grand nombre de variabilités. Ces dernières concernent les concepts liés au matériel et aux logiciels du domaine de la robotique. Par conséquent, le développement des applications de robotique est une tâche complexe. Non seulement, elle requiert la maîtrise des détails de bas niveau du matériel et du logiciel mais aussi le changement du matériel utilisé dans une application entraînerait la réécriture du code de celle-ci.

L'utilisation de l'ingénierie dirigée par les modèles dans ce contexte est une voie prometteuse pour (1) gérer les problèmes de dépendance de bas niveau des applications des détails de bas niveau à travers des modèles stables et (2) faciliter le développement des applications à travers une génération automatique de code vers des plateformes cibles. Les langages de modélisation spécifiques aux domaines mettent en oeuvre les techniques de l'ingénierie dirigée par les modèles afin de représenter les concepts du domaine et permettre aux experts de celui-ci de manipuler des concepts qu'ils ont l'habitude d'utiliser. Cependant, ces concepts ne sont pas suffisants pour représenter tous les aspects d'une application car ils très généraux. Il faudrait alors s'appuyer sur une démarche pour extraire des abstractions à partir de cas d'utilisations concrets et ainsi définir des abstractions ayant une sémantique opérationnelle.

Le travail de cette thèse s'articule autour de deux axes principaux. Le premier axe concerne la contribution à la conception d'un langage de modélisation spécifique au domaine de la robotique mobile (RobotML). Nous extrayons à partir d'une ontologie du domaine les concepts que les roboticiens ont l'habitude d'utiliser pour la définition de leurs applications. Ces concepts sont ensuite représentés à travers une interface graphique permettant la représentation de modèles afin d'assurer une facilité d'utilisation pour les utilisateurs de RobotML. On offre ainsi la possibilité aux roboticiens de représenter leurs scénarios dans des modèles stables et indépendants des plateformes cibles à travers des concepts qu'ils ont l'habitude de manipuler. Une génération de code automatique à partir de ces modèles est ensuite possible vers une ou plusieurs plateformes cibles. Cette contribution est validée par la mise en oeuvre d'un scénario aérien dans un environnement inconnu proposé par l'ONERA.

Le deuxième axe de cette thèse tente de définir une approche pour rendre les algorithmes résistants aux changements des détails de ba niveau. Notre approche prend en entrée la description d'une tâche de robotique et qui produit :

- un ensemble d'abstractions non algorithmiques représentant des requêtes sur l'environnment y compris le robot ou des actions de haut niveau ;
- un ensemble d'abstractions algorithmiques encapsulant un ensemble d'instructions permettant de réaliser une sous-tâche de la tâche étudiée ;
- un algorithme générique configurable défini en fonction de ces abstractions.

Ainsi, l'impact du changement du matériel et des stratégies définies dans les sous-tâches n'est pas très important. Il suffit d'adapter l'implantation de ces abstractions sans avoir à modifier l'algorithme générique. Cette approche est validée sur six variantes d'une famille d'algorithmes de navigation appelée Bug.

Abstract One of the challenges of robotics is to manage a large number of variability. The latter concerns the concepts related to hardware and software in the field of robotics. Therefore, the development of robotic applications is a complex task. Not only it requires mastery of low-level details of the hardware and software but also if we change the used hardware in an application, this would impact the code.
The use of model-driven engineering in this context is a promising way to (1) manage low-level dependency problems through stable models and (2) facilitate the development of applications through automatic code generation to target platforms . Domain Specific Modeling Languages implement the model driven engineering technologies to represent the domain concepts and enable experts to manipulate concepts they are used to use. However, these concepts are not sufficient to represent all aspects of an application because they are very general. We would then use an approach to extract abstractions from concrete use cases and thus define abstractions with an operational semantics .
The work of this thesis focuses on two main axes. The first concerns the contribution to the design of a domain specific modeling language for mobile robots (RobotML). We extract from a domain ontology concepts that roboticists have used to use to define their applications. These concepts are then represented through a graphical interface representation model to ensure ease of use for RobotML users. An automatic code generation from these models can then be performed to one or more target platforms. This contribution is enabled by setting implement an air scenario, in an unknown environment, proposed by ONERA.
The second focus of this thesis attempts to define an approach to make the algorithms resistant to the change of low-level details. Our approach takes as input a description of a task and produces robotic :
– a set of non-algorithmic abstractions representing queries on environnment (including robot) or high-level actions ;
– a set of algorithmic abstractions encapsulating a set of instructions to perform a sub-task of the studied task ;
– a generic configurable algorithm defined according to these abstractions.
Thus, the impact of changing hardware and strategies defined in the sub-tasks is not very important. Simply adapt the implementation of these abstractions without changing the generic algorithm. This approach is validated on six variants of a navigation algorithms family called Bug.