Conception et développement de contrôleurs de robots - Une méthodologie basée sur les composants logiciels

Robin Passama

Académie de Montpellier
Université Montpellier II
— Sciences et Techniques du Languedoc —

Thèse

présentée au Laboratoire d'Informatique, de Robotique
et de Microélectronique de Montpellier

Spécialités	:	**Informatique, Robotique**
Formation Doctorale	:	**Informatique**
École Doctorale	:	**Information, Structures, Systèmes**

Conception et développement de contrôleurs de robots

Une méthodologie basée sur les composants logiciels

par

Robin Passama

Soutenue le 30 Juin 2006, devant le jury composé de :

M. Jacques Malenfant, Professeur, LIP6, Université Pierre et Marie Curie, ... Rapporteur
M. Robert Valette, Directeur de recherche CNRS, LAAS, ... Rapporteur
M. René Zapata, Professeur, LIRMM, L'université Montpellier II, ... Président du Jury
M. Christophe Dony, Professeur, LIRMM, Université Montpellier II, ... Directeur de thèse
M. David Andreu, Maître de Conférence, LIRMM, Université Montpellier II, ... Co-directeur de thèse
M^{me} Thérèse Libourel, Maître de Conférence, LIRMM, Université Montpellier II, Co-directeur de thèse

Remerciements

Mes premiers remerciements iront à mes directeurs de thèse, David Andreu, Christophe Dony et Thérèse Libourel qui ont accepté de coencadrer cette thèse et qui m'ont témoigné leur soutien et leur confiance tout au long de ces années. J'exprime également ma sincère reconnaissance aux autres membres du jury, René Zapata, président, Jacques Malenfant et Robert Valette, rapporteurs, qui ont donné de leur temps et de leur sueur pour évaluer mon travail.

Je tiens à remercier Jean Privat, Luc Fabresse, Grégory Beurrier, José Baez et Jérome Chapelle avec qui j'ai partagé des discutions scientifiques gratifiantes et des moments de détente appréciés. Je n'oublie pas non plus ceux du LIRMM qui ont su rendre le cadre de travail très agréable : Lionel Lapierre, Mathieu Lafourcade, Alexis Criscuollo, Didier Schwab, Abdellah El Jalaoui, Mehdi Yousfi-Monod, Josette Durante et bien d'autres que je ne peux tous citer.

Je ne terminerai pas mes remerciements sans avoir une pensée forte pour Julie et mes parents, qui m'ont supporté, dans tous les sens du terme et en tous moments.

à Mireille et Lucien

Table des matières

Chapitre 1

Introduction

1.1 Contexte

La robotique a longtemps été confinée à la production sur des chaînes automatisées. Les premiers robots apparus étaient des "esclaves" mécaniques fixes, évoluant dans un espace clos (robots soudeurs sur une chaîne de montage par exemple). Leurs mouvements et interactions avec l'environnement physique étaient prédéfinis et répétitifs. Ces robots étaient des systèmes peu flexibles et peu adaptatifs. Depuis quelques décennies, la robotique s'est élargie à de nouveaux domaines comme, par exemple, l'exploration spatiale (Spirit et Opportunity de la NASA). Les robots ne sont plus de "simples" outils améliorant la productivité industrielle, ils deviennent de vrais machines intelligentes aidant l'homme dans la réalisation de tâches complexes, risquées, voire impossibles (cas de l'exploration spatiale). Cette nouvelle robotique, appelée robotique de service, est un enjeu majeur de l'industrie du futur. De nombreuses sociétés se positionnent d'ores et déjà sur ce marché en pleine croissance [52], dont les profits annoncés dans les années 2020 sont colossaux [48]. La robotique de service englobe les toutes dernières générations de robots qui apportent aux humains - de façon mobile et autonome (ou semi-autonome) - une aide concrète dans une multitude de circonstances. La robotique de service s'illustre, par exemple [56], par les recherches actuelles sur les robots mobiles d'assistance aux personnes âgées. Les robots intègrent alors, au-delà de leurs fonctionnalités classiques (déplacement en environnement encombré par exemple), un ensemble de services "annexes" liés à leur contexte d'utilisation : surveillance du rythme biologique, suivi des activités journalières, interactions avec l'environnement domotique, etc. Dans ce contexte de l'assistance aux personnes âgées, il se positionne également comme un relais entre l'individu et les intervenants extérieurs. Ces intervenants extérieurs peuvent, par exemple, être membres d'un corps médical (chargés du diagnostic d'anomalies sur le rythme biologique), ou peuvent être chargés de la surveillance à distance (ils peuvent lors d'une situation d'urgence détectée, comme une chute de la personne par exemple, téléopérer le robot à distance pour inspecter le logement). Globalement, la robotique de service réunit les robots agissant dans les secteurs suivants : santé (notamment les robots dits "interventionnels", c'est-à-dire pratiquant l'acte chirurgical), assistance aux personnes, divertissement (Aïbot de SONY), sécurité, maintenance, exploration spatiale, exploration et surveillance sous-marine, défense (drones et dernièrement "soldats" robots).

Bernard Espiau [47] a proposé une définition du concept de robot qui est appropriée à la robotique de service. Un robot est "*... une machine agissant physiquement sur son environnement en vue d'atteindre un objectif qui lui a été assigné. Cette machine est polyvalente et capable de s'adapter à certaines variations de ses conditions de fonctionnement. Un robot est doté de fonction de perception, de décision et d'action. Il possède des capacités de mouvement propres et peut entrer en interaction avec des objets de son environnement. Il a en outre, la faculté de coopérer à divers degrés avec l'homme.*" Cette définition souligne la complexité inhérente à la conception d'un robot de ser-

vice. Cette complexité est due à l'intégration de nombreuses fonctionnalités, allant de la perception à la réalisation d'actions, en passant par la prise de décision, et ce, à différents niveaux d'interaction avec l'homme et avec l'environnement. Un robot de service est vu comme un outil "intelligent", dont les capacités décisionnelles et opérationnelles sont sans commune mesure avec celles que les robots industriels avaient par le passé. Ceci s'explique par le fait qu'un robot de service doit évoluer dans un environnement partiellement inconnu et qu'il doit effectuer des tâches dont la complexité de réalisation nécessite des mécanismes de raisonnement, d'adaptation, de perception et d'action. Un robot de service doit pouvoir, dans certains contextes d'exécution, se passer au moins partiellement de l'intervention humaine dans ses prises de décisions. Par exemple, dans le cas d'une mission d'exploration spatiale, un robot doit être capable de réaliser des tâches complexes et de s'adapter à l'environnement extra-planétaire avec une intervention humaine limitée. En effet, l'homme ne peut ni le téléopérer directement à distance directement (délais de communication inter-planétaires trop grands), ni intervenir à ses côtés (environnement hostile et/ou inaccessible). La complexité de conception d'un robot de service provient donc de l'autonomie que ses développeurs veulent (en fonction du service qu'il doit rendre) et doivent (en fonction des contraintes environnementales) lui conférer.

Un robot est vu suivant deux axes : sa *partie opérative* et son *contrôleur*. La partie opérative est la partie matérielle (mécanique et instrumentation embarquée) qui lui permet d'interagir physiquement (perception et action) avec son environnement. Elle est essentielle, puisqu'elle intègre sa charge utile, i.e. l'instrumentation nécessaire à la réalisation des tâches qui lui sont confiées. En premier lieu, elle doit être en adéquation avec l'environnement dans lequel évolue le robot ; par exemple, le moyen de locomotion d'un robot sous-marin est en adéquation avec l'environnement liquide dans lequel il se déplace (e.g. torpille, robot-anguille). En second lieu, la partie opérative doit être en adéquation avec le service que le robot doit rendre. Dans le cas d'un robot d'exploration extra-planétaire, elle est par exemple constituée de moyens permettant : d'effectuer des prélèvements et des analyses de roches, d'évaluer la température et la pression, etc. Le contrôleur est la partie logicielle d'un robot, via laquelle il prend des décisions et contrôle sa partie opérative. Le contrôleur est également constitué des composants électroniques et liens de communication (solutions de traitement et de transmission de l'information) qui constituent le support nécessaire à l'exécution de ce logiciel. Ce support permet en particulier au contrôleur de s'interfacer avec la partie opérative, et impose des contraintes de déploiement au logiciel. Le logiciel encapsule l'ensemble des fonctionnalités qui mettent en œuvre la prise de décision, l'adaptation du robot à son contexte d'exécution, ainsi que le contrôle de sa partie opérative. Il est donc le cœur de l'autonomie d'un robot de service, sans lequel il est incapable de réaliser des tâches complexes.

Un contrôleur de robot est une application logicielle très complexe. Premièrement, les contrôleurs de robots, sont des systèmes logiciels dits "réactifs". Un système réactif contrôle un système matériel lui fournissant les informations à traiter (typiquement au travers de capteurs) et les moyens de le contrôler (typiquement au travers d'actionneurs). Un système réactif doit être capable de réagir dans un intervalle de temps tel que le système physique contrôlé n'a pas encore significativement évolué, cet intervalle de temps dépendant de la dynamique du système physique lui-même. Les contrôleurs de robots de service sont le plus souvent des systèmes embarqués, c'est-à-dire qu'ils s'exécutent sur des systèmes matériels "contraints", qui possèdent, par exemple, une autonomie énergétique propre. Enfin, les contrôleurs sont des systèmes hybrides, dans lesquels est gérée la cohésion entre un monde "continu" (le monde physique perçu et contrôlé) et un monde "discret" (la représentation du monde au sein du contrôleur). La conception de contrôleurs nécessite donc de prendre en considération chacun de ces aspects spécifiques.

Un contrôleur logiciel doit gérer la réalisation des services qu'un robot doit rendre pendant sa mission, en fonction des limitations imposées par sa partie opérative et par l'environnement dans lequel il évolue. Cela nécessite, en particulier, que le contrôleur gère les mécanismes d'adaptation (au sens large) du robot. De tels mécanismes permettent, plus précisemment de gérer :

- L'adaptation du robot à l'environnement dans lequel il évolue (e.g. présence d'obstacles statiques ou mobiles, passage d'un environnement structuré vers un environnement non structuré, etc.).
- L'adaptation des services proposés en fonction de ses utilisateurs humains (e.g. utilisateur expert ou non).
- L'adaptation du contrôle du robot en fonction des contraintes imposées par sa partie opérative (e.g. un mauvais fonctionnement mécanique ou énergétique).
- L'adaptation du contrôle en fonction de son contexte de mission (présence ou non d'opérateurs humains et/ou d'autres robots pouvant l'aider dans ses tâches, impliquant un possible changement de son mode de fonctionnement).
- L'adaptation du contrôle du robot en fonction de contraintes intrinsèques au contrôleur lui-même (e.g. temps de calcul, domaines de validité des lois de commande).

La gestion des mécanismes d'adaptation d'un robot, couplée à celle de mécanismes de perception, de décision et de contrôle, fait de la conception de contrôleurs de robots une tâche particulièrement ardue, à laquelle le travail présenté a été dédié.

1.2 Problématique

La conception d'un contrôleur de robot nécessite l'utilisation de techniques diverses et hétérogènes (e.g. intelligence artificielle, coopération multi-robots, interaction homme/machine, navigation, guidage, commande, estimation, etc.), s'appuyant sur des données également hétérogènes (e.g. carte de l'environnement, données capteurs d'origines diverses). Ceci pose alors le problème de l'intégration, dans un tout commun, de ces différentes techniques, afin de mettre en œuvre le système de prise décision et de contrôle d'un robot. La problématique d'intégration est d'essence logicielle et/ou technologique. Elle est d'*essence logicielle* pour ce qui concerne la mise en commun d'une multitude de traitements informatiques (e.g. planificateurs, mécanismes d'apprentissage, protocoles de coopération multi-robots ou protocoles d'interaction homme-machine, modes de fonctionnement, lois de commandes, etc.), afin qu'ils puissent être utilisés de façon combinée. L'intégration est d'*essence technologique* pour ce qui concerne l'utilisation de technologies diverses au sein d'un robot : technologies de communications sans-fil (e.g. Wi-Fi, Bluetooth, etc.), technologies d'acquisition de données (capteurs, caméra), technologies de traitement du signal (e.g. FPGA), etc. Le contrôleur doit, de fait, intégrer les fonctionnalités permettant l'utilisation de ces différentes technologies, afin de les faire interopérer. L'intégration de fonctionnalités hétérogènes suppose, en soi, que ces fonctionnalités puissent être manipulées indépendamment les unes des autres. Il est, en effet, difficile de concevoir des contrôleurs de façon monolithique. La problématique sous-jacente, est qu'il faut définir les fonctionnalités et les structures de données utilisées, et les organiser entre elles de façon cohérente, de manière à ce que le contrôleur soit conforme aux attentes des utilisateurs.

De plus, ces fonctionnalités doivent pouvoir être mises en relation les unes avec les autres de façon incrémentale et interchangeable, afin de pouvoir augmenter ou modifier les capacités d'un contrôleur (i.e. les services qu'il rend et/ou la façon dont il rend des services). C'est donc la capacité des contrôleurs à évoluer qui est primordiale dans le cadre de projets de développement robotiques. L'évolutivité des contrôleurs peut avoir des raisons de natures diverses. Elle peut être d'*ordre matériel*, lorsque par exemple l'instrumentation embarquée évolue. Elle est accentuée par l'essor technologique très fort dans le domaine, qui induit des mutations de technologies fréquentes. Cet état de fait impose le remplacement de certaines parties de l'instrumentation d'un robot, considérées comme obsolètes lorsque de nouveaux matériels (plus performants, plus robustes, moins coûteux, etc.) sont disponibles. L'évolutivité matérielle contribue aux exigences de modularité et d'évolutivité des contrôleurs, dont la partie matérielle est le support. L'évolutivité peut également être d'*ordre logiciel*, lorsque par exemple, de nouvelles versions de traitements informatiques sont disponibles. Cette évolutivité peut

être justifiée, par exemple, par la gestion de cas non prévus, ou pour des considérations d'optimisation. Enfin, l'évolutivité peut concerner les services rendus par les robots, ce qui couple souvent une évolutivité matérielle et logicielle (par exemple, la gestion de nouveaux dispositifs de perception et d'action dans un environnement domotique, permettant au robot de rendre de nouveaux services).

Un autre problème actuel est celui de la réutilisation des fonctionnalités et structures de données utilisées pour la conception de contrôleurs. La réutilisation est, à l'heure actuelle, essentielle afin de pouvoir construire des contrôleurs modulaires à partir de "briques logicielles" disponibles et fonctionnelles. Cela permet, en particulier, de réduire le temps et les coûts d'un projet de développement, puisque les développeurs n'ont qu'à créer les briques logicielles encapsulant les fonctionnalités spécifiques à leur système et à les assembler avec des briques logicielles existantes.

Les pratiques de développement actuelles ne prennent pas suffisamment en compte ces problèmes d'importance pour la conception de contrôleurs logiciels (intégration, évolutivité et réutilisation de fonctionalités hétérogènes). Ceci s'avère être un frein majeur à l'essor de la robotique de service. Sans une approche fournissant des solutions à ces problèmes, la gestion de la complexité intrinsèque à tout contrôleur est une tâche bien trop ardue. Prendre en compte ces problèmes comme le point central autour duquel baser le développement de contrôleurs de robots est devenu, depuis peu, un enjeu stratégique pour l'industrie robotique [50]. Plusieurs réseaux internationaux se sont formés, comme par exemple OROCOS [112], CLAWAR [41] et JAUS [53], afin de définir des architectures de contrôleur standards dans des domaines applicatifs différents. La standardisation est d'importance pour des considérations d'intégration et de réutilisation, car elle permet, en partie, la compatibilité des briques logicielles. Néanmoins, celle-ci ne semble pas suffisante pour régler l'ensemble des problèmes posés. En effet, la méthodologie et les outils utilisés pendant le développement sont aussi importants pour espérer régler l'ensemble des problèmes. Il est donc nécessaire de définir de nouvelles pratiques, adaptées au développement de contrôleurs de robots de services, qui intègrent à la fois une approche méthodologique et une approche pragmatique (outils utilisés). La définition de telles pratiques pose un ensemble de questions : comment favoriser la mutualisation des expertises des développeurs ? Comment organiser un contrôleur en terme de briques logicielles en interaction, qui chacune encapsule une ou plusieurs fonctionnalités d'un contrôleur ? Comment définir ces briques logicielles, quelles sont leurs propriétés ? Comment permettre et favoriser leur réutilisation et leur modularité ? Comment assurer que l'intégration de ces briques au sein d'un contrôleur soit valide (dans une certaine mesure) ? Comment réduire au mieux toutes les tâches pénibles et non-essentielles lors du développement, afin d'optimiser le rendement d'un projet ?

1.3 Objectif

Dans ce contexte et afin de donner des réponses adaptées aux questions précitées, l'objectif de cette thèse est de proposer une *méthodologie de conception et de développement* dédiée. Cette méthodologie doit donc favoriser l'évolutivité des contrôleurs et doit permettre la définition, l'intégration et la réutilisation de fonctionnalités sous forme de briques logicielles. Elle doit également proposer une pratique de développement qui facilite la mutualisation des compétences des développeurs.

Pour cela, nous avons étudié deux courants de recherche, qui nous semblent complémentaires pour le sujet traité : les architectures logicielles des contrôleurs de robots [71] et les approches basées sur les composants logiciels [114]. L'étude des architectures de contrôleurs, plus couramment appelées architectures de contrôle, nous permet d'avoir une vue d'ensemble sur les problématiques d'identification, d'intégration et d'organisation des blocs logiciels au sein des contrôleurs, ainsi que les solutions apportées. L'étude des méthodologies de développement logiciel à base de composants nous permet de connaître les approches proposées par le génie logiciel pour les problématiques de réutilisation de code, de lisibilité et de modularité des architectures logicielles.

Afin d'être validée, au moins conceptuellement, la méthodologie proposée doit être appliquée au développement d'un contrôleur de robot. L'exemple d'un robot manipulateur mobile retenu est suffisamment complet (hétérogénéité des lois de commande, des mécanismes d'adaptation, des modes de fonctionnement, de la partie opérative, etc.) pour évaluer la complexité de conception d'une architecture de contrôle. Il est ainsi possible de mieux comprendre les apports et limites de la méthodologie proposée.

1.4 Plan

La première partie de ce mémoire présente une étude bibliographique des différents domaines abordés pour répondre aux objectifs fixés. Dans cette partie, le chapitre 2 traite de l'étude des démarches de conception/développement proposées par le génie logiciel. Cette étude nous permet de définir les pratiques actuelles qui sont nécessaires ou utiles à la formalisation d'une méthodologie de développement logiciel de contrôleurs. A partir de cette formalisation, il est apparu nécessaire de comprendre les principes d'organisation fondamentaux des contrôleurs logiciels de robot. Le chapitre 3, présente l'étude bibliographique des architectures de contrôle proposées en robotique autonome, à partir desquelles nous tentons d'extraire des principes d'organisation des contrôleurs. A partir de la démarche de développement retenue, le besoin de concrétiser ces principes d'organisation nous ont amenés, dans le chapitre 4, à étudier les différentes propositions sur la notion de composant logiciel. Au delà d'une démarche de développement et de principes d'organisation, la méthodologie doit reposer sur des abstractions et des outils permettant d'une part la définition de briques logicielles réutilisables, et d'autre part leur intégration dans des architectures de contrôle. Le chapitre 5 dresse une réflexion globale autour de cette étude et définit notre positionnement.

La seconde partie de cette thèse définit avec précision la méthodologie de développement. A partir de la présentation générale faite au préalable, le chapitre 6 entre dans le détail de notre proposition d'un modèle d'organisation d'une architecture de contrôle. Ce modèle définit un ensemble de concepts à partir desquels est menée la phase d'analyse et il met en relation ces concepts avec différents types d'entités logicielles. Il sert de base à la mutualisation des expertises des développeurs, en fournissant un cadre pour l'intégration de fonctionnalités diverses (planification, commande, action, gestion des modes de fonctionnement, etc.). Ce modèle d'organisation devant être appliqué de façon concrète, le chapitre 7 présente le langage à composants servant à modéliser les architectures logicielles et à programmer les composants logiciels. Ce langage se focalise sur les notions de réutilisation, de modularité et d'évolutivité et prend en compte certains besoins inhérents au développement de contrôleurs. Le chapitre 8 présente l'environnement d'exécution de ce langage. A partir du langage à composants et du modèle d'organisation d'architectures de contrôle proposés, le chapitre 9 présente un exemple de développement d'une architecture de contrôle d'un robot manipulateur mobile, qui se conforme à la méthodologie. Le dernier chapitre présente le bilan de ce travail de thèse, à partir duquel nous orientons nos recherches et développements actuels et futurs.

Première partie

Etat de l'art

Chapitre 2

Démarches de développement

2.1 Introduction

Depuis plusieurs années, la recherche en informatique propose des solutions pour l'organisation de projets de développement logiciel. Plusieurs termes, de sens proches, ont été historiquement utilisés pour dénommer ces propositions ; nous utilisons le terme neutre de *démarches de développement* pour qualifier l'ensemble de ces propositions. Ces démarches de développement ont des portées souvent différentes, mais certains principes génériques existent. Le principe de base, autour duquel s'organise un projet de développement logiciel, est celui de *cycle de vie* d'un logiciel : c'est l'ensemble des étapes qui mènent de sa conceptualisation à sa mise en service. Le deuxième principe est celui de la *pratique de développement*, c'est-à-dire la façon dont les acteurs d'un projet d'ingénierie logicielle (que nous nommerons plus simplement développeurs) s'organisent entre eux afin de se répartir le travail de conceptualisation, de programmation, de test, etc. La pratique de développement est souvent définie, au sein d'un projet, en fonction du cycle de vie du logiciel. Le troisième principe, certainement le plus communément accepté de nos jours dans l'industrie du logiciel, est celui d'*objet*. Un objet est vu comme une unité de composition d'un logiciel et un logiciel est vu comme une composition d'objets en interaction. Une démarche de développement basée sur la notion d'objet apporte une bonne lisibilité d'un logiciel, c'est-à-dire qu'elle permet à une personne de bien appréhender la structure et les fonctionnalités essentielles d'un logiciel. Cette lisibilité qu'apporte la vision d'objet provient essentiellement de la mise en correspondance des concepts manipulés par les développeurs avec les entités abstraites ou informatiques constituant le logiciel.

Ce chapitre présente quelques démarches de développement proposées par la recherche et l'industrie. Ces démarches sont génériques, ou dédiées au domaine de l'automatique. Les principes génériques sus-cités y sont déclinés de manières différentes. La conclusion de ce chapitre propose une réflexion sur les différentes démarches de développement présentées.

2.2 UML et les Processus de Conception

Depuis de nombreuses années UML [26] [65] est devenu le langage de modélisation standard utilisé massivement dans l'industrie du logiciel. Il a été défini à partir d'un consensus autour d'anciennes propositions de langages de modélisation orientés objet. UML est un langage de modélisation sur lequel s'appuient de nombreuses démarches de développement. Les démarches de développement les plus génériques existant à l'heure actuelle sont appelées *processus de conception*. Ces processus visent à définir des pratiques de développement de logiciels autour de leurs cycles de vie. Il existe à l'heure actuelle deux grandes catégories de processus de conception :

- Les processus dits "itératifs" sont, à l'heure actuelle, les plus courants dans l'industrie du logiciel. Ces processus sont "rigoureux", dans le sens où ils mettent en avant la documentation

de la spécification du logiciel et le suivi d'un projet via de nombreux points de contrôle du travail effectué (conformité logiciel/spécification). Suivre un tel processus consiste à découper un projet en extrayant des sous-ensembles de fonctionnalités qui seront traitées itérativement les unes après les autres. Chaque itération traite de l'ensemble du cycle de vie de la fonctionnalité traitée : analyse, conception, programmation et tests. Une fois une itération menée à terme, l'itération suivante démarre afin de traiter une autre fonctionnalité du projet, ou à améliorer/corriger les fonctionnalités existantes. Ces processus peuvent prendre plusieurs formes, mais globalement ils suivent tous ce schéma. L'avantage de suivre un processus itératif est de permettre aux membres d'un projet de se rendre compte au plus tôt des problèmes techniques ou de modélisation rencontrés lors du développement d'un logiciel et d'ainsi, pouvoir les traiter indépendamment lors de chaque itération, ou entre itérations. Dans ce cadre, l'utilisation des différents diagrammes UML est quasi-généralisée pour documenter le développement.

– Les processus dits "agiles", beaucoup plus récents, gagnent en "popularité" comparés aux processus itératifs. "Agile" est un terme désignant un ensemble de valeurs et de principes définis par le *Manifesto for Agile Software Development* [19], à l'image de celle nommée *eXtreme Programming*. Le principe des processus "agiles" est de centrer le développement autour des personnes, alors que les méthodes ou outils utilisés sont des préoccupations de second ordre. Dans les processus agiles la documentation et la modélisation ne sont pas centrales. Ces processus proposent, plutôt, une entrée en phase de développement/test au plus tôt et reposent souvent sur l'utilisation de méthodes de refactoring/rétroconception. Les processus agiles sont peu "rigoureux", dans le sens où ils imposent peu de documentation et peu de points de contrôle tout au long d'un projet. Dans ce cadre, UML est surtout utilisé comme une notation standard pour créer une première esquisse du modèle d'un logiciel, voire parfois comme un support de programmation (pour créer le squelette de la structure du logiciel). L'avantage de tels processus est qu'ils permettent de se rendre compte au plus tôt de la complexité technique de certaines parties d'un projet et par là-même de pouvoir y réagir au plus vite.

Le processus présenté par la suite est un processus standard, nommé RUP, qui est représentatif de la catégorie des processus itératifs. Notre préoccupation étant la conception de contrôleur de robots, il nous semble difficile d'appliquer, dans ce contexte, des processus aussi peu structurés que les processus agiles. En effet les applications robotiques demandent, plus que certaines applications classiques en informatique, une grande rigueur pendant le développement. Le Rational Unified Process (RUP) est un processus de développement logiciel [73], qui vise à fournir un ensemble de méthodes permettant de transformer les besoins des utilisateurs en logiciels. Le RUP est un processus générique, adaptable en fonction des besoins associés à chaque projet. Dans le RUP, UML est le langage de modélisation par défaut, car il contient suffisamment de diagrammes et d'extensions pour permettre la modélisation des très nombreuses préoccupations des producteurs de logiciels.

Le RUP oriente tout le travail d'un projet autour du cycle de vie du logiciel et en particulier, autour du cyle de vie de l'architecture logicielle. Cette dernière est considérée comme une vue d'ensemble structurée (i.e. le modèle) d'une application. L'architecture est décomposée en un ensemble d'entités que nous nommerons par défaut objets. Le RUP est un processus itératif, où le cycle de vie est décomposé en un ensemble d'itérations. Chaque itération correspond à un ajout ou une amélioration de fonctionnalités, ce qui équivaut au développement d'un ensemble d'objets d'une architecture, ou à l'amélioration des objets existants. Chaque itération est découpée en un ensemble de phases (cf. fig. 2.1). Les processus itératifs sont incrémentaux entre les phases d'une itération, c'est-à-dire que des informations sont ajoutées à l'architecture logicielle en cours de définition, à chaque nouvelle phase. Pendant chacune de ces phases, un sous-ensemble de diagrammes proposés par UML est utilisé pour documenter un projet :

– La phase de modélisation des besoins de l'utilisateur est documentée en utilisant les diagrammes de cas d'utilisation. Elle consiste à décrire les attentes des utilisateurs d'un logiciel.

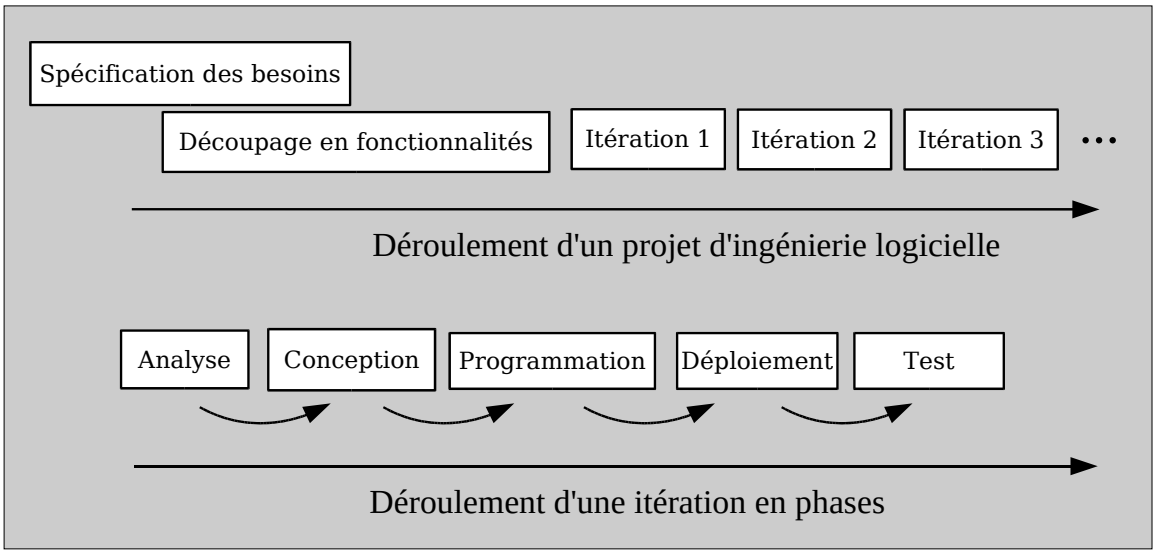


FIG. 2.1 – Processus itératif de type RUP

- La phase d'analyse consiste à détailler les cas d'utilisation et permet de faire une première répartition des objets au sein d'une architecture. Les diagrammes de classes sont utilisés pour représenter les concepts "métier" et leur relations. Les diagrammes de séquence sont utilisés pour définir plus précisemment les interactions entre certains objets de l'architecture.
- La phase de conception consiste à définir l'architecture logicielle. Cette architecture est, en particulier, définie par des modèles de classes raffinant ceux issus de la phase d'analyse. Lors de cette phase de nombreux diagrammes statiques ou dynamiques d'UML sont utilisés, pour décrire le comportement associé à chaque classe (diagrammes statecharts), des exemples d'organisations d'objets (diagrammes d'objets), ou des exemples d'interactions typiques entre objets (diagrammes de séquence).
- La phase de programmation consiste à écrire l'implantation des différentes classes en fonction des modèles issus de la phase précédente. A ce stade, UML est utilisé la plupart du temps comme support de la documentation et plus rarement, comme base à partir de laquelle générer le code squelette des applications (certains Ateliers de Génie Logiciel permettent, cependant, de générer du code à partir de différents diagrammes UML).
- La phase de déploiement consiste à définir et effectuer le placement du code logiciel d'une application sur le système physique sur lequel elle est exécutée. Cette phase peut être documentée par le biais de diagrammes UML, de paquetages, de composants et de déploiement.
- La phase de test consiste à effectuer des procédures de test qui permettent, en cas de succès, de valider les cas d'utilisation du logiciel. Lors de cette phase, les diagrammes UML de l'application, tant structurels que comportementaux, peuvent servir de base à la définition des procédures de test.

Le RUP définit un ensemble de rôles joués par des intervenants humains (nommés acteurs) participant au processus de conception. Chaque rôle correspond à une tâche précise à réaliser dans le cadre d'une itération. Un même acteur peut potentiellement jouer plusieurs rôles et peut donc intervenir à différentes phases d'une même itération. Nous simplifions la proposition du RUP en ne donnant que les rôles d'acteurs qui nous semblent majeurs, plusieurs sous-rôles pouvant être définis par leur spécialisation.

- Un *architecte* est un acteur responsable de la définition de l'architecture logicielle d'une application. Son travail travail consiste à coordonner celui des autres acteurs du processus, autour du modèle de conception et des cas d'utilisation de l'application.
- Un *programmeur* a pour tâche de produire, documenter et archiver les implantations (code source et/ou compilé) des classes spécifiées par les architectes. Il peut raffiner les diagrammes de conception, créer les diagrammes de composants et peut définir et/ou s'appuyer sur les diagrammes de séquences, lors de la phase de test de l'application.
- Un *déployeur* est responsable du déploiement de l'application sur un ensemble de sites d'exécution. Pour ceci le déployeur a plusieurs tâches à accomplir. Il doit en premier lieu définir l'insfrastructure matérielle d'accueil de l'application et le placement du code logiciel sur cette infrastructure. Il peut documenter son travail via des diagrammes paquetage, de composant et de déploiement. En second lieu, il est responsable du déploiement physique des paquetages sur les sites d'exécution, en fonction des informations contenues dans les diagrammes de déploiement.
- Une fois les applications en exécution, elle requièrent, dans plusieurs cas, d'être administrées, en particulier celles devant être en fonctionnement 24h/24h ou celles susceptibles d'être étendues (en terme de fonctionnalités ou d'utilisateurs). Un *administrateur* est responsable de la supervision d'une application à l'exécution : il peut agir sur l'architecture de l'application afin d'en reconfigurer la topologie, ajouter ou supprimer des fonctionnalités, observer les éventuelles fautes ou pannes etc. Pour cela, l'*administrateur* doit disposer d'un outil adéquat lui permettant de visualiser et de modifier l'architecture tout en répercutant ces modifications sur l'application elle-même.

L'utilisation d'UML est particulièrement adaptée pour suivre le processus générique du RUP, car UML lui-même propose un ensemble de diagrammes variés permettant de documenter toutes les phases du cycle de vie d'une application logicielle. De plus UML possède un ensemble d'extensions permettant de modéliser des aspects particuliers de certaines classes de systèmes, comme par exemple les systèmes temps-réel, qui sont d'importance pour nous. Les travaux menés sur la proposition UML/PNO [93] [94] visent à promouvoir l'utilisation des réseaux de Petri (RdP) dans RT-UML pour modéliser les aspects dynamiques d'une application temps-réel. L'utilisation des RdP permet en particulier la validation/vérification, à l'aide de la logique linéaire [116], des comportements logiques et temporels d'objets ou de composition d'un ensemble d'objets. Elle permet la vérification de la cohérence entre des modèles dynamiques d'UML (e.g. vérification de la cohérence entre plusieurs diagrammes de séquence et/ou d'activité).

L'utilisation d'UML a travers des processus itératifs tels que le RUP, a permis, progressivement, la définition d'un ensemble de modèles objets, appelés patrons (patterns en anglais), dont le but est de permettre la réutilisation de solutions efficaces et bien connues. Ces solutions peuvent être spécifiques ou non à un domaine d'application, pour la modélisation d'une architecture logicielle ou pour la programmation de ses classes. Les patrons les plus connus sont les *Design Patterns* [57], qui définissent des solutions génériques pour la conception d'applications à base d'objets. Il existe, à l'heure actuelle, une garnde panoplie de patrons différents, spécifiques à chaque phase d'un processus logiciel itératif. Ils sont utilisés massivement dans l'industrie et la recherche, et se sont imposés comme une solution très efficace en terme de réutilisation de savoir-faire.

De par la puissance d'expression d'UML et de ses extensions, et de par la précision du RUP, les développeurs de logiciels disposent aujourd'hui d'outils et de méthodes de gestion de projet qui permettent de documenter et de superviser tout le cycle de vie d'un projet. De plus, via la souplesse d'utilisation d'UML (choix des diagrammes) et du RUP (choix des phases, des acteurs), il est possible d'adapter le processus de développement aux besoins d'un projet. Enfin, l'utilisation des patrons étant déjà très répandue, les usagers d'UML possèdent une base de savoir-faire réutilisable conséquente. Pour ce qui est du domaine d'application visé, l'utilisation de RT-UML, qui regroupe un ensemble

d'extensions pour la gestion de la qualité de service, l'ordonnancement, la tolérance aux fautes, etc., semble appropriée en ce qui concerne les phases d'analyse et de conception d'un système temps-réel. UML commence à susciter un intérêt certain dans la communauté robotique, certains travaux s'appuyant sur UML pour la conception de contrôleurs temps-réels [119]

2.3 La méthodologie HOOD/PNO

La méthodologie HOOD/PNO [92] est une démarche de développement proposée par le LAAS. C'est une extension de la méthodologie HOOD, dédiée à la conception et au développement de systèmes temps-réels et plus précisément aux procédés de commande industriels. Elle vise à produire une décomposition des architectures logicielles en objets dont les comportements concurrents et asynchrones sont décrits par le biais des réseaux de Petri à Objets [100]. Si HOOD était essentiellement utile lors des phases de modélisation des architectures logicielles, la méthodologie HOOD/PNO se base sur un cycle de vie complet, allant de l'analyse d'un système jusqu'à sa programmation.

La méthodologie HOOD/PNO propose une vision à la fois ascendante et descendante de la modélisation d'un système logiciel. A toutes les phases du développement logiciel, l'architecture logicielle est décrite à base d'objets. Chaque objet est, soit une entité passive (son unique activité est entièrement contrôlée par les flots de contrôle des objets l'utilisant), soit une entité active (qui a plusieurs activités parallèles). Dans ce dernier cas, le comportement parallèle est décrit par le biais des réseaux de Petri à objets (RdPO). Chaque objet possède : une interface offerte qui liste les opérations que d'autres objets peuvent appeler, une interface requise qui liste les opérations que l'objet appelle, un corps constitué d'attributs et d'opérations. Le corps d'un objet (passif ou actif) peut également contenir d'autres objets. Il existe deux types de relations entre objets. La relation *use* indique qu'un objet en utilise un autre, c'est-à-dire que son interface requise fait référence à une ou plusieurs des opérations définies dans cet autre objet. La relation *include* indique qu'un objet est contenu dans le corps d'un autre objet, ce qui signifie que l'objet incluant est défini, entre autre, à partir de l'objet inclus. HOOD/PNO repose sur l'identification des objets et leur décomposition (relation *include*), si besoin est, en un ensemble d'objets inclus reliés entre eux par des relations *use*. La notion de classes d'objets est également présente afin de permettre la factorisation des propriétés communes des objets, dans un but de réutilisation.

A partir de cette structure de base qu'est l'objet, HOOD/PNO repose sur un cycle de vie d'un logiciel décomposé en itération, chaque itération étant constituée de plusieurs phases. Ces phases sont les mêmes que celles présentées dans le processus RUP, mais elles ont été plus spécifiquement définies pour la description d'application de commande de procédés industriels :

- La phase d'analyse se fait suivant une approche ascendante. Elle consiste à identifier les différents objets physiques participant au procédé industriel commandé et à caractériser leurs interactions. Par exemple, pour un procédé de contrôle d'un poste de distribution automatique de carburant, l'analyse débouche sur l'identification de trois objets constituants le système physique commandé : le `poste-carburant` représente le système dans sa globalité. Il contient (relation *include*) les deux autres objets que sont la `caisse` et la `pompe`. L'objet `caisse` utilise l'objet `pompe` (relation *use* pour l'appel de ses opérations de livraison) afin qu'il livre la quantité souhaitée par le client. Les objets physiques identifiés et représentés sous la forme d'objets sont ensuite regroupés sous la forme de classe en fonction de leurs propriétés communes. Chacune de ces classes est associée à un RdPO qui décrit le comportement général des objets instances, c'est-à-dire la façon dont les activités de ces objets s'enchaînent, ou se déroulent en parallèle, en fonction des messages reçus (appels d'opérations).
- La phase de conception est faite suivant une approche ascendante et/ou descendante. L'approche descendante consiste à définir le comportement global d'un système (RdPO), puis de

CLASSE POSTE_CARBURANT

(1)
- **Adaptée à tous types de carburants,**
- **Encaissement automatique de billets et de pièces,**
- **Sélection du type de carburant par l'utilisateur,**
- **Ticket de livraison automatique, ...**

(2)
- **Régulation automatique du débit,**
- **Détection de plein,**
- **… ,**
- **Conçue pour implémentation avec langage de bas niveau ou de haut niveau**

Spécification de la classe

DESCRIPTION DETAILLÉE

Texte en langage naturel donnant toutes les caractéristiques de la classe.

La classe Poste_Carburant fournit un objet capable de piloter une pompe industrielle dotée d'un automate d'encaissement et de livraison détaillée de carburant. Elle comprend un objet caisse pour la commande et l'affichage et un objet pompe proprement dit avec livraison automatique d'une quantité de même régulation de débit ...

PARAMETRES
- Types (3)
- Variables (3)
- Opérations (3)

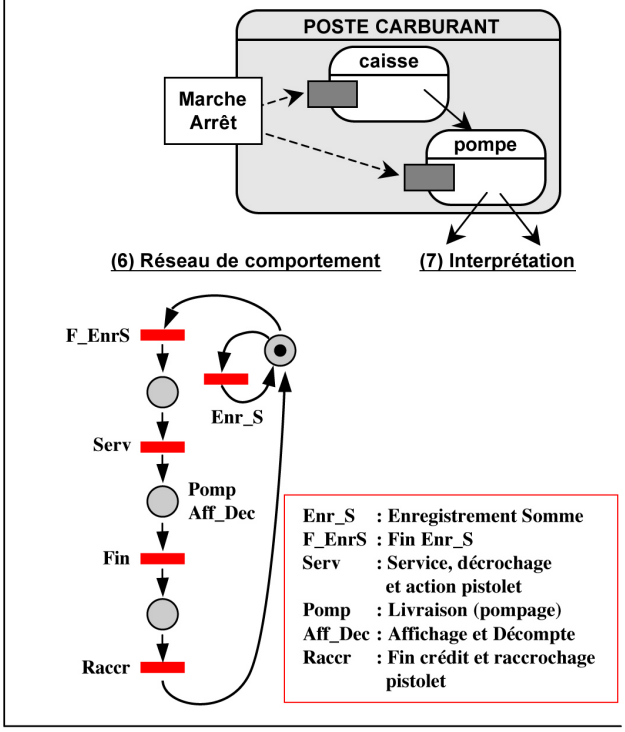
INTERFACE OFFERTE
- Types (4)
- Constantes (4)
- Opérations (4)
- Exceptions (4)

INTERFACE REQUISE
- Types (5)
- Constantes (5)
- Opérations (5)
- Exceptions (5)

STRUCTURE DE CONTRÔLE
- Réseau de Pétri (voir graphe) (6)
- Interprétation associée (7)

CORPS
- Objets (8)
- Types (9)
- Constantes (9)
- Opérations (9)
- Exceptions (9)
- OPERATIONS (10)
  - STRUCTURE DE CONTRÔLE
  - Opérations requises
  - Exceptions propagées (11)
  - Exceptions gérées (12)

Représentation graphique



(1) *Besoins généraux utilisateurs*
(2) *Besoins plus spécifiques (ingénieur logiciel)*
(3) *Paramètres formels (effectifs pour les instances)*
(4) *Description textuelle (like Ada) de l'interface offerte*
(5) *Pour chaque objet requis, description des principales caractéritiques de l'objet serveur : types, constantes, opérations et exceptions propagées par le serveur*
(8) *Champ vide si l'objet est terminal*
(9) *Description détaillée pour les objets terminaux ou de l'interface des objets inclus pour les objets non terminaux*
(10) *Pour chaque opération offerte, description textuelle et/ou réseau de Petri machine à bras*
(11) *Exceptions levées et propagées lors de l'exécution de l'opération*
(12) *Exceptions gérées localement (propagées ou non)*

FIG. 2.2 – Analyse d'un système de distribution de carburant avec HOOD/PNO [92]

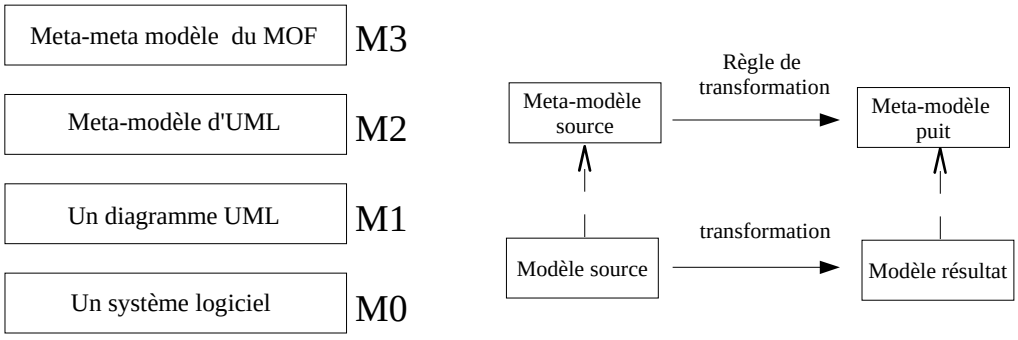
découper ce comportement, afin d'en faire émerger différents objets qui encapsulent chacun une sous-partie du comportement global. Ce découpage est guidé par les documents issus de la phase d'analyse (classes). Il est également guidé par l'utilisation de technique d'analyse des réseaux de Petri. Les auteurs proposent un ensemble de techniques qui vont permettre de décomposer l'objet système le plus global en un ensemble d'objets contenus et ce, de façon récursive, afin d'obtenir une représentation desagrégée du système. L'approche ascendante consiste à décire le comportement (RdPO) de chaque objet identifié dans la phase d'analyse, en fonction du comportement défini pour leur classe. Ces objets sont ensuite composés via une relation *use* à l'intérieur d'un objet les incluant. Le comportement de ce nouvel objet est alors déduit de la composition des comportements (i.e. des RdPO) des objets inclus. La possibilité offerte par HOOD/PNO d'aborder la modélisation par décomposition ou composition d'objets fournit aux développeurs les moyens de manipuler leurs modèles objets à différents niveaux d'abstraction, en fonction des comportements qu'ils sont capables de modéliser à un niveau donné.
- HOOD/PNO propose un ensemble de pratiques de programmation afin de transformer les modèles de la phase de conception en code Ada. L'avantage majeur de cette proposition est de permettre une programmation qui limite les erreurs humaines de traduction des modèles.

La méthodologie HOOD/PNO est intéressante puisqu'elle fournit des solutions amenant une plus grande rigueur dans le suivi d'un projet logiciel. Elle utilise en particulier, de façon spécifique, le formalisme des RdPO. Celui-ci qui permet de vérifier certaines propriétés comportementales. Il permet également d'utiliser des mécanismes d'analyse spécifiques comme support de la modélisation. Toutefois HOOD/PNO semble complexe à manipuler dans sa globalité. Son côté extrêmement "rigoureux", avec l'utilisation massive des propriétés des RdPO, limite fortement son utilisation par des non-spécialistes de ce formalisme de modélisation. L'utilisation d'outils logiciels permettant d'automatiser les mécanismes utilisés pendant la modélisation et permettant de générer le code logiciel équivalent, semble nécessaire. La méthodologie HOOD/PNO permet de mettre en exergue les freins majeurs à l'utilisation d'une méthodologie de développement très "rigoureuse". Le premier frein est que peu de gens sont capables de l'assimiler dans sa globalité tant sa complexité est grande. Le second frein est que la tâche d'écriture des documents de spécification est très lourde, ce qui se révèle d'autant plus contraignant si ces documents ne peuvent être exploités afin de générer du code logiciel.

2.4 L'approche MDA

Depuis quelques années l'OMG concentre ses activités de recherches sur une nouvelle méthodologie de développement orientée autour de la modélisation et de la meta-modélisation, nommée Model Driven Architecture (MDA) [64] [33]. Celle-ci centre une grande partie de la production des applications logicielles autour de la production de modèles. Les modèles ne servent plus uniquement à écrire les documents de spécifications d'un projet d'ingénierie logicielle, mais ils sont massivement utilisés lors de la phase de programmation. La naissance de MDA est issue d'un constat : le nombre et la variété des plates-formes logicielles de développement (par exemple le nombre de langages de programmation) interdit à ce jour l'émergence d'une technologie de développement unique, qui n'existera certainement jamais. De ce constat, l'OMG a établi un besoin primordial pour les développeurs de concevoir leurs applications à un niveau d'abstraction supérieur, indépendant, autant que possible, de toute plate-forme technologique. La modélisation et la meta-modélisation permettent de capitaliser la façon dont sont construites les applications en les décrivant sous forme de modèles, eux-mêmes décrits dans des langages de modélisation. La méthodologie MDA repose sur des techniques permettant de raffiner et d'adapter les modèles à différentes plates-formes technologiques. L'automatisation partielle des techniques de production de modèles est le centre des recherches actuellement menées dans le cadre de MDA.

La compréhension de MDA nécessite, a priori, la compréhension des notions de modèles et meta-



(a) Pile du MOF (b) Transformation de modèles

FIG. 2.3 – Pile de meta-niveau et transformation de modèles dans MDA

modèles. Selon J. Bézivin [32], un modèle est une représentation abstraite d'un système (logiciel dans le cas qui nous intéresse), qui devrait être plus simple que le système considéré. Un modèle peut être utilisé pour répondre à un certain nombre de questions sur ce système. Un meta-modèle est un modèle qui définit un ensemble de concepts importants et de relations entre ces concepts, à partir desquel on peut décrire des modèles. En d'autres termes le meta-modèle définit la terminologie à utiliser pour définir des modèles et tout modèle est conforme à un meta-modèle. Par exemple, le langage UML possède un meta-modèle qui définit les différents diagrammes utilisables au sein d'UML ainsi que les entités et relations modélisables (objets, classes, composants, associations, collaborations, etc.) dans chaque diagramme UML. Dans le but de donner un cadre cohérent à la meta-modélisation, l'OMG a créé le Meta Object Facility (MOF) [91], une notation qui repose sur un meta-meta-modèle. Le meta-meta-modèle du MOF permet la définition de meta-données et de relations entre ces meta-données, à partir desquelles des meta-modèles peuvent être décrits. Par exemple, le meta-modèle d'UML est conforme au meta-meta-modèle du MOF. Ainsi l'OMG peut imposer le MOF comme le langage "pivot" à partir duquel sont décrits les meta-modèles. L'utilisation du MOF au sein de l'approche MDA, est basée sur une pile de meta-niveaux qui contient quatre niveaux distincts (cf. fig. 2.3 (a)) :

- Le niveau M0 est celui qui contient conceptuellement le monde "réel" qui est modélisé, c'est-à-dire une application logicielle dans le contexte du développement d'applications.
- Le niveau M1 est le niveau contenant les modèles qui décrivent des applications. Ce niveau correspond par exemple à un modèle UML d'une application.
- Le niveau M2 contient la description de la structure et de la sémantique des modèles. Ce niveau correspond par exemple au meta-modèle d'UML.
- Le niveau M3 contient la description de la structure et de la sémantique des meta-modèles. Ce niveau correspond au meta-meta-modèle du MOF. Pour éviter une infinité de meta-niveaux l'OMG propose un niveau M3 méta-circulaire, c'est-à-dire qui suffit à se décrire lui-même.

L'essence de MDA repose dans l'utilisation de modèles catégorisés en deux groupes distincts : les modèles indépendants des plates-formes, nommés PIM (pour *Plateform Independant Models*) et les modèles prenant en compte les spécificités des plates-formes technologiques, nommés PSM (pour *Plateform Specific Models*). Les modèles PIM permettent aux développeurs de s'abstraire complètement des aspects techniques liés à l'implantation d'un système logiciel, c'est-à-dire de ce

que l'OMG appelle *plate-forme*. Une plate-forme peut être un middleware à composants logiciels (e.g. CORBA) ou un langage de programmation (e.g. C++), voire un système d'exploitation (e.g. linux). Un modèle PIM décrit l'application uniquement en terme de concepts "métiers", il est donc portable à travers différentes plates-formes puisque conçu indépendamment de celles-ci. Un des principes de MDA est de raffiner un modèle PIM afin de créer de nouveaux modèles PIM. Ceci permet aux développeurs de raisonner à différents niveaux d'abstraction. Un modèle PSM décrit une application en prenant en considération des problématiques de programmation et/ou de déploiement. Les modèles PSM peuvent également être raffinés dans de nouveaux modèles PSM lorsque la technologie sous-jacente le permet ou l'impose (e.g. par exemple un raffinement d'un modèle PSM CORBA vers un modèle PSM CORBA/C++ dans lequel la programmation des comportements des composants logiciels est prise en compte). Le cœur de la méthodologie MDA est de proposer les techniques et outils permettant de raffiner les modèles PIM et PSM entre eux ainsi que de générer des modèles PSM à partir de modèles PIM. Ceci permet aux développeurs d'adapter leurs modèles "métier" (PIM) à des technologies de développement diverses (PSM). Inversement, le but de MDA est aussi de permettre de générer des PIM à partir de PSM (abstraction des spécificités des plates-formes). Pour gérer la génération de modèles, MDA centre tout le processus d'écriture de modèles autour du mécanisme de *transformation de modèles*.

Une transformation de modèles, pour être automatisable, doit être décrite à partir de règles exprimées sur les deux meta-modèles utilisés (cf. fig. 2.3(b)) : celui ayant permis de décrire le modèle source et celui ayant permis de décrire le modèle résultat. A partir d'un ensemble de règles de transformation, tous les modèles décrits via le meta-modèle source peuvent être transformés en modèles décrits via le meta-modèle résultat. Afin d'écrire ces règles de transformation de modèles, l'OMG préconise d'exprimer les meta-modèles sources et puits dans un langage de meta-modélisation commun et standard tel que le MOF. Les règles de transformation de modèles sont décrites comme des opérations entre des éléments des meta-modèles source et résultat. L'OMG tente actuellement d'établir un langage d'écriture de règles de transformation standard. Une grande partie de la complexité de la mise en œuvre de MDA réside à l'heure actuelle dans la façon de représenter et d'automatiser ces règles de transformation [32].

Via le principe de transformation de modèle, ainsi que ceux de PIM et PSM, MDA permet d'aborder de façon originale le principe de séparation des préoccupations (*SoC* pour *Separation of Concerns* en anglais) [78]. La *SoC* a influencé de nombreux travaux de recherche comme la programmation par Aspects [76] par exemple. Dans le principe de la *SoC*, les préoccupations des développeurs sont considérées commes les points autour desquels organiser et décomposer une application. Le découpage des différentes préoccupations peut être guidé par des besoins fonctionnels (découpage en objets métiers), techniques (séparation des propriétés propres à un intergiciel) ou humains (place d'un acteur du processus dans le cycle de vie). Ce découpage vise avant tout à fournir à chaque développeur, une vision compréhensible et manipulable de l'application, qui contient uniquement les détails le concernant dans le cycle de vie de l'application. Chaque développeur travaille sur un modèle qui traite d'une préoccupation précise. Il manipule pour cela les entités du meta-modèle correspondant à la préoccupation qu'il traite. La séparation des préoccupations doit être accompagnée d'un processus d'intégration des différentes préoccupations en un tout commun, appelé processus de *tissage* (*weaving* en anglais). Le tissage reste, à l'heure actuelle, difficile à contrôler ou à mettre en œuvre lorsque plusieurs préoccupations sont composées. La transformation des modèles permet de réaliser le tissage des préoccupations par étapes, en raffinant un modèle d'un niveau d'abstraction à un autre et en fusionnant des modèles d'un même niveau d'abstraction. Les préoccupations techniques (liées à la programmation ou au déploiement par exemple) sont gérées au moment de la transformation des modèles PIM en modèles PSM (ou par raffinement de modèles PSM).

MDA est selon nous une approche complémentaire aux processus itératifs, car elle s'attache davantage à résoudre les problèmes de réutilisation des modèles et de génération de "code", que les

problèmes d'organisation du travail et de documentation autour du cycle de vie d'une application. Le but sous-jacent à l'utilisation de MDA est de déporter la plus grande partie du travail de programmation des applications vers celui de définition de modèles, laissant la partie exécution et administration à la charge de plates-formes diverses et variées. Ainsi, MDA s'adapte bien à l'évolution de l'industrie du logiciel, se plaçant comme une méthodologie standard permettant de s'abstraire au maximum de l'utilisation des nombreuses plates-formes proposées sur le marché. Il est plus que probable que MDA soit également fortement utilisée dans les processus "agiles" car elle favorise la rétro-conception (par exemple passage d'un PSM vers un PIM) et la génération rapide de code. En effet, MDA peut être utilisée avec les langages UML et MOF, déjà fortement outillés pour la production semi-automatisée de code, pour l'analyse et pour la rétro-conception de modèles. Il existe par exemple la possibilité, via l'utilisation de profils, de raffiner un modèle UML standard pour chaque plate-forme cible (e.g. C++, Java, CORBA IDL etc.) puis de générer le code adéquat pour la plateforme cible considérée via des outils propres à cette plate-forme. De plus, le MOF est associé à un ensemble d'outils logiciels permettant de produire des environnements logiciels qui supportent l'utilisation de meta-modèles. L'approche pragmatique de MDA dans sa mise en œuvre de la *SoC*, nous laisse à penser que cette méthodologie permettra de mieux comparer les atouts et défauts des plates-formes proposées par le marché. En effet les meta-modèles PSM définis pour chaque plate-forme technologique, vont permettre chacun de gérer un ensemble de préoccupations techniques. La possibilité de générer très vite du code d'une même application pour différentes plates-formes favorisera la comparaison des préoccupations techniques que sont capable de gérer ces plates-formes. Néanmoins, la réussite de MDA repose essentiellement sur les réponses apportées par la recherche sur l'ensemble des problèmes posés par la transformation de modèles et l'écriture de règles de transformation.

2.5 Réflexions sur les démarches de développement logiciel

Cette étude nous amène à faire un ensemble de constats sur les principes fondamentaux des démarches de développement.

2.5.1 Objets

Tout d'abord, dans toutes ces démarches proposées, l'objet (ou la classe d'objets) est l'entité de base à partir de laquelle sont décrits les modèles ou programmées les applications. La vision objet permet de décomposer l'architecture d'une application en terme d'entités assumant chacune un ensemble de propriétés de l'application et interagissant entre elles. L'apport majeur des méthodologies "objet" est la possibilité offerte aux développeurs, de mettre en adéquation les concepts "métiers" qu'ils manipulent, avec des entités de modélisation qui permettent de représenter leurs problèmes. Dans ces méthodologies, l'architecture logicielle reflète, au moins en partie, la façon dont les concepts sont structurés et interagissent. Ainsi ces méthodologies ont prouvé, de par leur utilisation massive dans l'industrie, qu'elles permettent d'améliorer significativement la lisibilité des applications. Cette lisibilité facilite la compréhension de l'architecture logicielle, ce qui contribue à faciliter sa modification ou la réutilisation de ses parties.

D'autre part, une méthodologie "objet" donne la possibilité de mettre en correspondance les entités modélisées avec les entités de programmation qui permettent leur mise en œuvre. Plus généralement, les objets sont utilisés dans toutes les phases du cycle de vie d'un logiciel, depuis la modélisation jusqu'à la programmation et l'exécution, ce qui amène une certaine traçabilité du cycle de vie d'une application.

2.5.2 Cycle de vie : Phases, Itérations et Acteurs

L'une des préoccupations majeures des processus itératifs est de fournir une solution générique au suivi d'un projet de développement logiciel. En ce sens les processus itératifs proposent un découpage du cycle de vie d'une application en différentes itérations pendant lesquelles les différentes fonctionnalités d'un logiciel sont développées. Chaque itération peut elle-même être découpée en un ensemble de phases distinctes : analyse, modélisation, programmation et exécution (administration/test). L'organisation des différents acteurs d'un projet logiciel se fait donc autour d'un cycle de vie décomposé en itérations et phases. Un acteur est responsable de la réalisation d'un ensemble de tâches pendant une itération, chaque tâche dépendant d'une des phases qui compose l'itération.

Ce découpage du cycle de vie d'un logiciel en itérations et en phases, semble suffisamment simple à comprendre et à mettre en œuvre pour être retenu comme une "constante" d'un processus de conception/développement. La différence entre les deux catégories de processus de conception est essentiellement de l'ordre de la pratique. Les processus "agiles" favorisent un passage plus rapide vers les phases de programmation et de test sans formaliser les phases précédentes (i.e. pas de description de modèles comme avec le RUP), dans le but de rendre chaque itération plus courte. Dans les processus itératifs "formels", une itération est un passage d'une phase de description abstraite vers une phase de description plus concrète, à l'inverse des processus agiles qui favorisent un cheminement inverse par rétro-conception. Dans tous les cas, les acteurs participants à un projet réalisent les mêmes phases : analyse du problème, modélisation de la solution, programmation du logiciel implantant la solution, déploiement, exécution et tests du logiciel. La différence tient dans la manière de réaliser ces phases. Les processus "itératifs" sont plus "rigoureux", car ils proposent aux acteurs de décrire des documents de spécification pour chaque phase.

La simplicité d'un processus de conception reste, à notre avis, la condition sine qua none à son utilisation sur des projets réels, en particulier si ces projets n'impliquent pas que des spécialistes en informatique (ce qui est la cas dans notre problématique). Le principe d'itération et le découpage d'une itération en différentes phases sont suffisamment simples à appréhender pour être le cœur d'une démarche de conception/développement.

2.5.3 Raffinement et intégration

Une des problématiques majeures et actuelles est d'organiser le travail au sein d'une itération entre les différents acteurs qui œuvrent à sa réalisation. Afin d'organiser ce travail, il faut proposer aux acteurs un cadre qui leur permet d'exprimer les différents points de vue qu'ils ont d'une application en fonction de leur spécialité. Il faut également leur donner la possibilité de raffiner et d'intégrer ces points de vue dans un tout commun. Si les langages de modélisation actuels permettent d'exprimer ces points de vues, par exemple à travers différents diagrammes UML, leur raffinement et leur intégration reste un défi. La puissance d'expression qu'atteint actuellement MOF 2.0 et UML 2.0 et les techniques d'intégration et raffinement proposées dans MDA font, qu'à l'heure actuelle, l'utilisation des modèles devient incontournable dans l'ingénierie du logiciel. MDA propose en particulier de décrire différents modèles d'une même application puis de raffiner et intégrer incrémentalement ces modèles jusqu'à obtenir un modèle détaillé de la structure et du comportement du logiciel. La possibilité de raffinement et d'intégration des modèles via *transformation* donne un cadre méthodologique cohérent pour la séparation des différentes préoccupations, naissant lors du cycle de vie d'une application. Il devient ainsi possible de raisonner sur une application à plusieurs niveaux d'abstraction et sous différents points de vue. Puisque dans MDA, les modèles sont utilisés pour automatiser la production du code des applications, cette méthodologie permet, indirectement, d'intégrer différentes parties de code logiciel.

Les principes de raffinement et d'intégration de différents points de vue sont essentiels pour le développement de logiciels. Il n'est pas dit que MDA sera la solution unique pour mettre en œuvre

ces mécanismes, d'autres propositions pouvant émerger dans le futur. Même si la transformation de modèles reste, à nos yeux, la référence actuelle pour exprimer ces mécanismes, elle nous semble lourde en terme d'efforts que doit consacrer le développeur (écriture des meta-modèles et des règles de transformation, écriture des modèles) et qui plus est difficile à appliquer tant que les recherches autour de la transformation de modèles n'auront pas apporté de solutions tangibles sur l'ensemble des problèmes qu'elle pose. Des solutions alternatives existent comme par exemple celle proposée par HOOD/PNO : le raffinement y est issu de l'application d'une pratique de développement dédiée qui permet de raffiner les modèles (analyse vers conception) et qui permet de traduire directement les modèles dans du code logiciel. Cette pratique pourrait être automatisée afin de ne laisser aux développeurs que la tâche d'écrire le code relatif aux aspects non couverts par le modèle. Cette approche propose donc de mettre en correspondance directe les entités manipulées au niveau des modèles avec celles manipulées au niveau des logiciels. L'approche HOOD/PNO est très spécifique comparée à MDA qui est généraliste. Néanmoins, elle propose un raffinement beaucoup plus simple à prendre en main, en particulier pour le passage entre modèles et code (i.e. modèle exécutable).

Résumé

Une démarche de développement :

- est organisée suivant un processus de conception/développement « itératif »
  - répartition du travail en définissant différentes fonctionnalités à développer
  - une ou plusieurs itérations par fonctionnalité à développer
  - une itération décomposée en phases (analyse, conception, programmation, déploiement, test)
  - utilisation de la notion d'objet (lisibilité, traçabilité)

- est guidée par :
  - la production de modèles
  - le raffinement de modèles
  - l'intégration des préoccupations

- est outillée de manière à :
  - faciliter la production de code
  - permettre une entrée rapide en phase de test

Chapitre 3

Architectures des contrôleurs

3.1 Introduction

Depuis plusieurs années les roboticiens s'intéressent à la conception d'architectures de contrôleurs de robots autonomes. Selon F. Ingrand [71], l'architecture d'un contrôleur (plus communément appelée *architecture de contrôle*) définit les entités logicielles et matérielles utilisées pour réaliser le système de prise de décision d'un robot, ainsi que les modalités d'interaction entre ces entités. Les architectures de contrôle sont catégorisées en trois grandes familles, en fonction du modèle d'organisation utilisé lors de leur conception : les architectures délibératives, les architectures comportementales et les architectures mixtes. Dans ce chapitre, nous nous intéressons aux modèles d'organisation des architectures de contrôle, c'est-à-dire aux modèles qui définissent les principes d'organisation des briques logicielles au sein de ces architectures.

Le modèle d'organisation des architectures **délibératives**, également appelées architectures hiérarchisées (cf. fig. 3.1), centre la conception sur le système décisionnel. Ces architectures sont organisées, dans la plupart des propositions, en plusieurs couches (également appelées niveaux) hiérarchisées [59] [3] [2]. Une couche communique uniquement avec la couche directement inférieure et la couche directement supérieure. Ces architectures comportent typiquement trois couches :

- La couche *fonctionnelle* contient des modules de perception qui sont en charge de la transformation des données numériques, provenant des capteurs, en données symboliques. Elle contient également des modules de génération de trajectoires et des modules d'asservissement, qui sont responsables de la transformation des données provenant du niveau supérieur, en données numériques applicables aux actionneurs.
- La couche *exécutive* est en charge de la supervision des tâches du robot (e.g. "se diriger vers la source de chaleur la plus proche"). Elle contient pour cela des modules qui dirigent l'exécution des modules de la couche *fonctionnelle*.
- La couche *décisionnelle* est en charge des mécanismes de planification (i.e. création et supervision de plans). Elle gère la façon dont le robot va enchaîner différentes tâches afin de réaliser chaque objectif de sa mission. Les mécanismes de décision de plus haut niveau, appelés traditionellement mécanismes de *délibération* sont les mécanismes centraux dans ces architectures [117], basés sur des techniques d'intelligence artificielle.

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

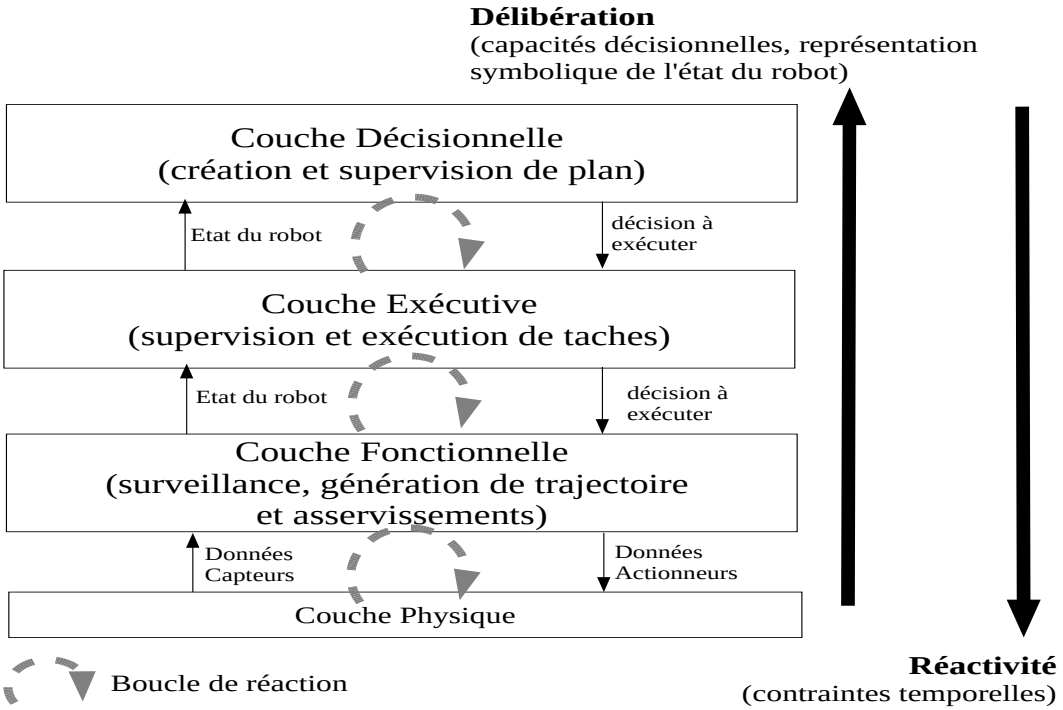


FIG. 3.1 – Modèle d'organisation des architectures délibératives

applique alors, en fonction de cette décision, les asservissements adéquats (i.e. activation des modules d'asservissement qu'elle contient).

Le principal problème des contrôleurs conçus avec de telles architectures, vient de leur manque de réactivité, c'est-à-dire la capacité de réagir en temps voulu à des modifications dans l'environnement. Ce manque de réactivité provient de l'organisation en couches : le transfert de l'information doit obligatoirement se faire en traversant toutes les couches intermédiaires. Ainsi, une adaptation décidée dans la couche décisionnelle (ex : modification du plan courant) se base sur des informations provenant exlcusivement de la couche exécutive, qui elles-mêmes proviennent du traitement d'informations provenant de la couche fonctionnelle. La décision d'adaptation est alors propagée vers la couche fonctionnelle en traversant les couches intermédiaires. Avec cette forme d'organisation, nous constatons que les traitements réalisés dans les couches intermédiaires induisent des latences d'autant plus importantes que le nombre de couches (et donc de traitements) est grand. Nous constatons également que dans de nombreux cas, traiter l'adaptation dans la couche décisionnelle se révèle très coûteux en temps de réaction, en particulier à cause de la lourdeur des mécanismes de délibération. En effet, l'occurence d'événements temporellement imprévisibles induit souvent une replanification au niveau de la couche décisionnelle. Si le calcul de la planification est sans garantie de temps borné (en adéquation avec la dynamique du robot), le robot doit alors être arrêté. Dans les architectures délibératives, le comportement global du robot est défini en fonction de hiérarchie de mécanismes de décision/contrôle : ce comportement est donc maîtrisé et compréhensible par l'homme, mais l'approche manque globalement de réactivité.

Le modèle d'organisation des architectures **comportementales** (cf. fig. 3.2), aussi appelées architectures réactives, centre la conception autour des réactions réflexes du robot. Ces architectures sont constituées d'un ensemble d'entités qui correspondent chacune à un comportement réactif du robot. Ces comportements réactifs s'exécutent en parallèle et le plus souvent de façon périodique. Un com-

portement réactif [13] perçoit un ensemble de stimuli (vu comme un vecteur d'état constitué à partir des données capteurs les plus à jour) et génère immédiatement une réaction du robot (vue comme un vecteur de commande). La réactivité d'un contrôleur ayant une architecture comportementale est "maximale" puisque la moindre information provenant des capteurs est immédiatement répercutée sur la commande générée. Chaque comportement réactif étant défini de manière indépendante, le problème majeur de conception de ces architectures est lié à la façon de composer ces comportements. En effet, une architecture de contrôle est définie à partir de plusieurs comportements réactifs (i.e. plusieurs stratégies de réactions), qui possèdent leur finalité propre pour le contrôle du robot. Or, ces comportements réactifs peuvent générer, au même moment, des commandes qui sont contradictoires. Par exemple, si un comportement réactif fait que le robot est attiré par une source de chaleur et qu'un autre fait que le robot s'éloigne quand la température devient trop importante, les commandes générées peuvent être opposées à un moment donné, l'une tendant à éloigner et l'autre à rapprocher le robot de la source de chaleur. Afin de résoudre ce problème, les architectures comportementales intègrent une *règle d'arbitrage*. Cette *règle d'arbitrage* est chargée de récupérer périodiquement l'ensemble des vecteurs de commande générés par les comportements réactifs et d'en déduire l'unique vecteur de commande envoyé aux actionneurs. L'enjeu de l'approche comportementale est de décrire une règle d'arbitrage qui permette au robot d'adopter un comportement global cohérent en fonction de la tâche qu'il a à accomplir, ce qui s'avère souvent particulièrement compliqué. La façon d'intégrer une telle règle à une architecture comportementale peut varier suivant l'approche retenue. Les travaux initiaux de Brooks [28], amènent une solution à l'intégration de cette règle d'arbitrage, avec les *subsumption architectures*. Les comportements réactifs y sont conceptuellement organisés en "niveaux de compétences", classifiés selon une vision hiérarchique (du plus prioritaire vers le moins prioritaire). Les comportements ayant un "niveau de compétence" donné sont capables, à n'importe quel moment, de "prendre le contrôle du robot", c'est-à-dire que les comportements réactifs des "niveaux de compétences" moins prioritaires ne seront pas pris en compte (les vecteurs de commande qu'ils génèrent ne seront pas appliqués aux actionneurs). Cette approche initiale a évolué vers des propositions qui intègrent de façon explicite (sous la forme d'une entité) les règles d'arbitrage dans les architectures. L'architecture DAMN [96] propose, par exemple, l'utilisation d'un protocole de vote dirigé par un arbitre. Cet arbitre reçoit, à chaque pas de temps, le vecteur généré par chaque comportement réactif et applique une règle d'arbitrage qui somme et pondère les vecteurs de commande générés. L'écriture des règles d'arbitrage se revèle très difficile dès lors que les vecteurs d'entrées sont grands et que les comportements réactifs s'exécutant en parallèle sont nombreux. En effet, raisonner sur les vecteurs d'entrée et de sortie d'un robot revient à raisonner dans un monde entièrement numérique, ce qui est extrêmement compliqué dès lors que les vecteurs deviennent grands. De plus, lorsque le contexte du robot (environnement, état et objectif courant) évolue, comme c'est le cas dès lors que le robot a une autonomie de mouvement et d'action dans l'environnement, la règle d'arbitrage doit pouvoir évoluer en conséquence. Définir les différents contextes possibles, à ce niveau d'abstraction (i.e. dans un monde essentiellement numérique), s'avère très difficile si le robot atteint une certaine complexité. Des réponses à ces deux problèmes (écriture et évolution des règles d'arbitrage) ont également été proposées. Dans l'architecture DAMN, l'arbitre se charge lui-même de faire évoluer les poids de la règle de sommation et de pondération, en fonction des votes de certains comportements réactifs. Brooks [29] a proposé l'utilisation des réseaux de neurones pour résoudre ce problème. Ainsi les règles d'arbitrage se définissent partiellement et évoluent automatiquement, à l'exécution. Dans les architectures comportementales, l'"intelligence" du robot provient de l'émergence d'un comportement global du robot [30], issue d'une composition plus ou moins maîtrisée des comportements réactifs, en fonction des interactions que le robot a eu avec son environnement. Le comportement global du robot est fortement réactif et adaptatif, mais difficilement compréhensible et maîtrisable par l'homme : il est particulièrement difficile pour l'homme de contrôler le déroulement de la mission du robot.

Architectures comportementales et architectures délibératives ont des défauts et des avantages

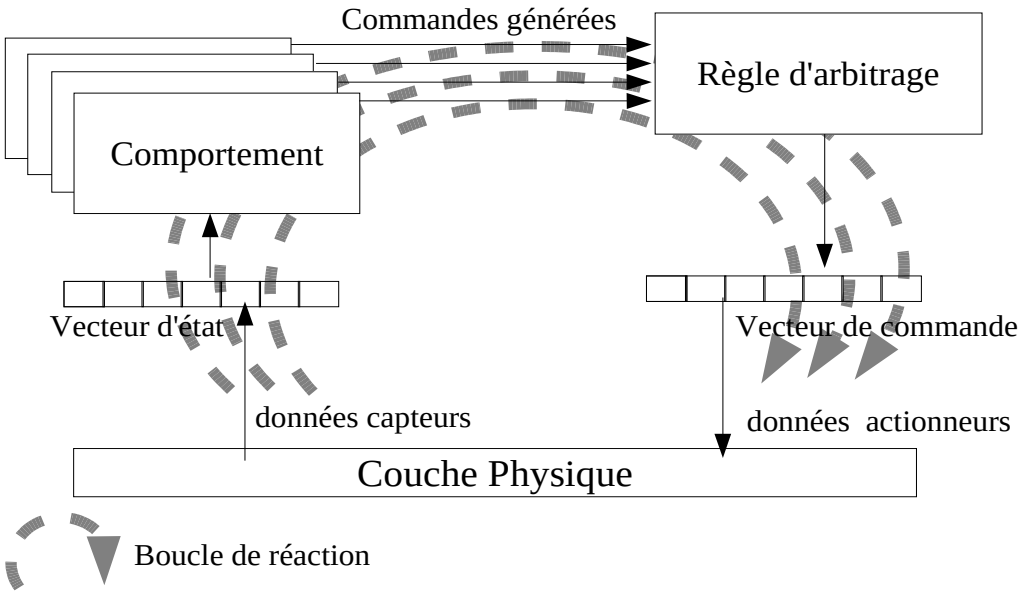


FIG. 3.2 – Modèle d'organisation des architectures comportementales

respectifs et complémentaires. Elles sont sous-tendues par des "philosophies" de développement très différentes. Les architectures délibératives sont construites avec la volonté de contrôler la façon dont le robot réalise sa mission, quitte à sacrifier certaines de ses capacités d'adaptation et sa réactivité. Les architectures comportementales sont construites avec la volonté de laisser la mission émerger de son interaction avec l'environnement, quitte à sacrifier (en partie) le contrôle que l'homme a sur lui pendant le déroulement de sa mission. Ce constat a amené à la définition d'une troisième catégorie d'architectures : les *architectures mixtes* (parfois appelées *architectures hybrides*). Les modèles d'organisation sous-jacents aux architectures mixtes concilient la prise de décision à différentes couches (afin de la rendre compréhensible et contrôlable par l'homme) avec l'optimisation de la réactivité en prenant et appliquant les décisions d'adaptation à la couche adéquate. La présentation de différents modèles d'organisation d'architectures de contrôle mixtes, faite dans les sections suivantes, est représentative des propositions existantes, sans pour autant être exhaustive (IDEA [87], 3T [25], TCA [103], MAAM [82] ne sont pas présentées en détail).

3.2 Architecture du LAAS

Le LAAS (Laboratoire d'Analyse et d'Architecture des Systèmes) propose un environnement logiciel constitué d'outils pour modéliser, programmer, exécuter et dans une certaine mesure valider, des architectures de contrôle [1]. Cet environnement s'appuie sur un modèle (cf. fig. 3.3) qui définit un découpage d'une architecture de contrôle en trois couches (appelées niveaux dans cette proposition) :

– Le niveau *fonctionnel* constitue l'interface entre les entités des couches supérieures et la partie physique du système. Il est le siège des fonctions de base du système : fonctions sensori-motrices, fonctions d'asservissement (e.g. navigation), fonctions de traitement (e.g. planificateur de trajectoire, segmentation d'image). Toute fonction est encapsulée dans un *module* (non donné aux entités logicielles constituant la couche fonctionnelle), généré par l'outil GenoM [51]. Un module offre un ensemble de services accessibles via des requêtes asynchrones (permettant de

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

Le modèle d'organisation des architectures de contrôle proposé par le LAAS est basé sur les principes d'organisation des architectures délibératives hiérarchisées. En effet, les mécanismes de délibération (planification et supervision de plans) du niveau décisionnel y sont centraux. A l'image des dernières architectures délibératives hiérarchisées telle que 4D/RCS [2], il existe des mécanismes d'adaptation à différents endroits : dans la hiérarchie des modules fonctionnels (adaptation des modules fonctionnels utilisés par un module fonctionnel plus global), dans l'exécutif procédural (adaptation, dans une certaine mesure, des modules fonctionnels utilisés pour réaliser une action) et dans le planificateur exécutif procédural (reformulation sur échec d'action). La décision d'adaptation du comportement du robot peut être prise aux plus hauts niveaux décisionnels (planification), mais peuvent également résulter d'actions réflexes programmées dans des modules de la couche *fonctionnelle*. L'autre constat est que l'information ne fait pas de saut entre les couches, par exemple directement de la couche fonctionnelle vers la couche décisionnelle, elle passe automatiquement par la couche de contrôle d'exécution. On remarque néanmoins que l'interaction entre la couche décisionnelle et la couche fonctionnelle est quasi-directe, la couche de contrôle d'exécution n'existant que pour assurer sûreté et robustesse du contrôle. On pourrait donc voir, in fine, ce modèle d'organisation comme étant à deux couches. Tout ceci rend difficile le classement de ce modèle d'organisation dans la catégorie des architectures délibératives, ou dans celle des architectures mixtes.

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

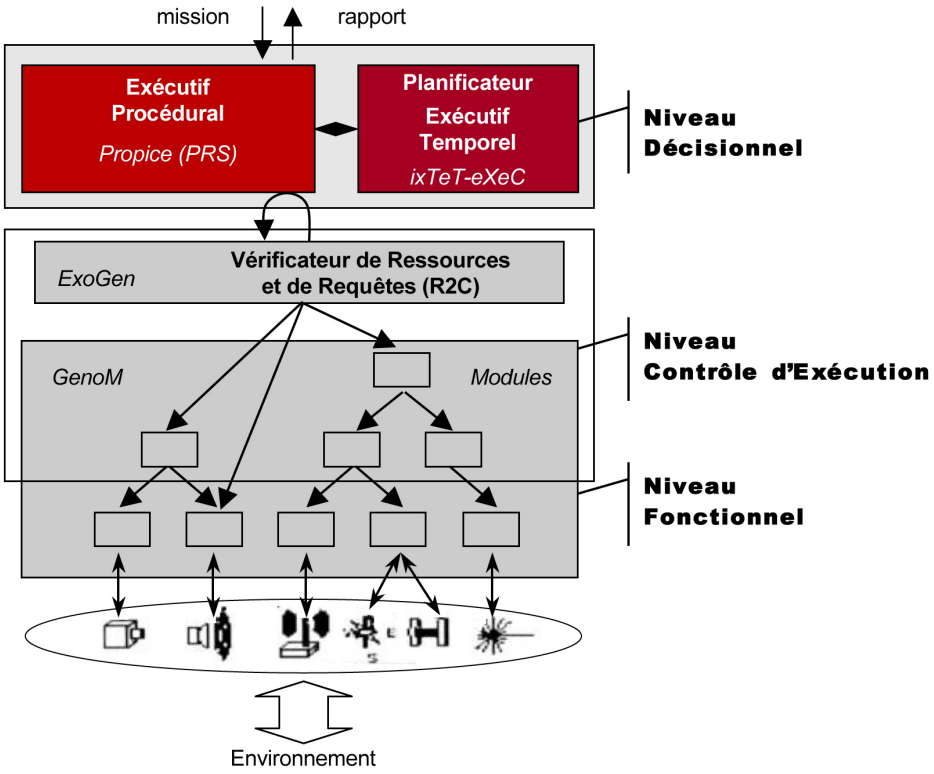


FIG. 3.3 – Modèle d'organisation des architectures de contrôle du LAAS [1]

nuancés en prenant en compte le fait que la couche fonctionnelle peut être organisée en une hiérarchie de modules en interaction. Identifier les différents types de modules (e.g. asservissement, perception, navigation, etc.) et leurs interactions typiques permettrait de mieux comprendre la façon dont le contrôle est organisé.

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

- Le planificateur/exécutif temporel IXTET-EXEC permet de générer et exécuter des plans temporels ; il est capable de prendre en compte dynamiquement de nouveaux objectifs de missions et les échecs d'exécution. Il est basé sur des technique de résolution de problèmes de contraintes (CSP).
- L'exécutif procédural PRS/Propice permet de superviser la réalisation d'actions ; il est réactif aux événements provenant de l'opérateur ou de la couche inférieure.
- Le contrôleur d'exécution R2C reconstitue dynamiquement l'état des ressources du robot (en fonction des requêtes et bilans d'exécution qu'il contrôle) ; il permet de filtrer les requêtes en fonction de cet état et d'un modèle formel des états acceptés ou non.
- L'outil Genom [51] permet de générer les modules du niveau fonctionnel à partir d'une description formelle (basée sur des machines à état finis) du comportement des modules.

Notons que la proposition du LAAS intègre l'utilisation de méthodes de spécification et de validation [44]. Les auteurs ont proposé, en particulier, l'utilisation des réseaux de Petri pour l'analyse et la validation des plans de l'exécutif procédural PRS.

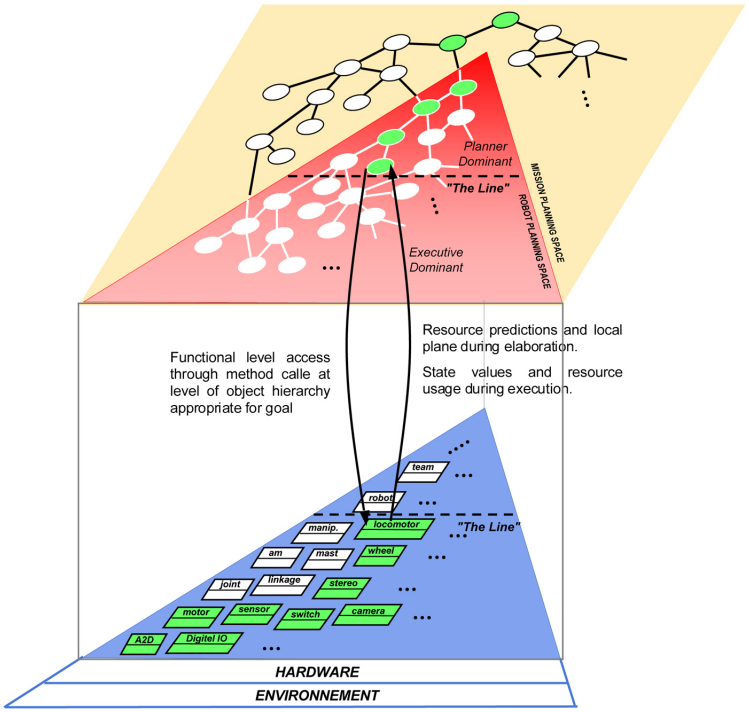


FIG. 3.4 – Modèle d'organisation d'architectures CLARATy [88]

3.3 CLARATy

CLARATy (Coupled Layer ARchitecture for Robotic Autonomy) [123] [124] est un modèle d'architecture mixte proposé par le NASA Jet Propulsion Laboratory et l'université Cannergie Mellon de Pittsburg. CLARATy est vue comme la proposition du futur pour le développement de contrôleurs de robots à la NASA. Dans le modèle CLARATy, la décomposition d'une architecture de contrôle se fait en deux couches : la couche fonctionnelle et la couche décisionnelle.

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

- les *Data Structure Classes* permettent de représenter, manipuler et stocker les données sous une forme standard, base nécessaire à une réutilisation efficace. Ces classes peuvent être génériques ou spécifiques à un domaine d'application. Les bits, les vecteurs, les matrices, les images, les messages, les quaternions, sont des exemples de tels objets.
- les *Generic Physical Classes* permettent de représenter des entités physiques, par exemple le locomoteur, la camera, la roue, le manipulateur. Chacune de ces classes possède une interface qui définit les fonctionnalités de l'élément matériel physique correspondant. Par exemple, l'interface de la classe locomoteur contient des opérations permettant de déplacer le véhicule en spécifiant le chemin à suivre et sa vitesse. La spécialisation de ces classes permet classiquement de leur

ajouter des fonctionnalités, d'autre part elle permet d'adapter les concepts physiques abstraits à un système robotique particulier, permettant ainsi de gérer spécifiquement le matériel utilisés dans le robot (les auteurs parlent alors de *Specialized Physical Classes*).
- les *Generic Functional Classes* sont des classes qui représentent un algorithme générique. Une telle classe utilise, pour définir son contenu, un ensemble de *Generic Physical Classes*. Par exemple, la classe navigateur permet de contrôler le déplacement d'un robot dans une zone (évaluation du terrain, calcul du chemin, sélection des actions à entreprendre, etc.). La description détaillée du navigateur peut être trouvée dans [120]. Ces classes peuvent être spécialisées afin d'adapter un algorithme à un système robotique particulier, ce qui sert essentiellement à optimiser des algorithmes en fonction du matériel considéré.

La couche décisionnelle est en charge de la gestion de l'"intelligence" du robot et est spécifique à chaque robot. Elle est utile à la planification et à l'exécution de plans. Elle permet de transformer les objectifs de haut niveau d'abstraction, reçus par le robot, en une série d'objectifs ordonnés suivant un ensemble de contraintes connues et selon l'état du système. Afin de réaliser des objectifs, la couche décisionnelle interagit avec les objets de la couche fonctionnelle, à travers un protocole client/serveur lui permettant : de connaître l'état des ressources physiques du robot à un moment donné, de contrôler les asservissements du robot, de demander l'exécution de traitements spécifiques (e.g. calcul de trajectoire). Cete couche est décrite suivant un *Goal Net*, qui représente la décomposition des buts de haut niveau d'abstraction en sous-buts. Chaque but ou sous-but peut être décomposé jusqu'à arriver aux buts feuilles qui accèdent directement à la couche fonctionnelle. Cette décomposition du niveau décisionnel sous forme d'un *Goal Net* reste conceptuelle, différents outils pouvant servir à sa mise en œuvre.

L'utilisation de l'approche objet permet aux auteurs de CLARATy de définir un ensemble de Frameworks orientés objets dédiés à la construction d'applications robotiques (par exemple, un framework de *Generic Physical Classes*[88]). CLARATy a pour principal atout la réutilisation des classes de la couche fonctionnelle, les auteurs annonçant un taux de réutilisation du code de l'ordre de 80% à travers les différents projets de la NASA utilisant CLARATy. L'apport de CLARATy vient de la possibilité de représenter explicitement les éléments matériels constituant le robot, et les fonctionnalités qui y sont rattachées, sous forme d'objets. L'approche objet est une solution intuitive pour représenter la partie physique d'un robot. Elle permet de décrire ou programmer le contrôle d'un robot à différents niveaux d'abstraction, du plus générique au plus spécialisé en fonction du matériel. Elle permet la mise en œuvre progressive de Frameworks spécialisés (locomotion, manipulation, vision, etc.) assurant une très forte réutilisation. D'un autre côté, la description du contenu du niveau décisionnel demeure, à notre connaissance, relativement "floue", les auteurs n'imposant pas de formalisme ou d'outil spécifiques. La proposition reste néanmoins intéressante, car les auteurs suggèrent que la couche décisionnelle peut elle-même être décomposée en sous-mécanismes de prise de décision, suivant une organisation "récursive". En conclusion, le modèle d'organisation de CLARATy peut être qualifié de mixte, puisqu'il décompose les architectures en deux couches en interaction directe : l'information transite directement d'une couche à l'autre et n'a donc pas plusieurs couches intermédiaires à franchir, ce qui permet d'optimiser la réactivité. Notons, pour conclure, que, tout comme l'approche du LAAS, CLARATy reste essentiellement inspirée des architectures délibératives classiques.

3.4 ORCCAD

L'INRIA (Institut National de Recherche en Informatique et Automatique) propose une technologie dédiée à la conception et à l'intégration de systèmes robotiques, nommée ORCCAD (Open Robot Computer Aided Design) [27] [104]. ORCCAD est suportée par un ensemble d'outils logiciels permettant de programmer et de valider des architectures de contrôle.

ORCCAD repose, implicitement, sur un modèle qui définit l'organisation d'une architecture de contrôle en deux couches : une couche *commande* et une couche *application*. La couche *commande* contient un ensemble d'entités logicielles nommées *Robot-Tasks* (RTs) qui représentent chacune une action robotique élémentaire, basée sur une loi de commande. La couche *application* est décrite par le biais d'un ensemble de *Robot-Procedures* (RPs) qui représentent des actions robotiques plus complexes, issues de la composition de RTs (et éventuellement de RPs). Les RPs sont en charge de la gestion d'une mission et de l'adaptation au contexte du robot. Nous présentons par la suite ces deux sortes d'entités.

Une *Robot-Task* (RT) est définie de la façon suivante : *a RT is defined as the parametrized specification of an elementary control law, i.e. the activation of a control scheme structurally invariant along the task duration, and of a logical behavior associated with a set of signals which may occur just before, during and just after the task execution.* La spécification du comportement logique est faite en définissant les signaux émis et reçus par la RT. Les signaux sont catégorisés de la façon suivante :

- les préconditions, sont les signaux dont l'occurrence est requise pour démarrer la tâche. Elles peuvent être exprimées par une expression logique ou par une mesure sur la valeur des capteurs.
- les exceptions sont les signaux notifiant l'occurence d'une situation d'erreur durant l'exécution de la tâche. Les exceptions sont de plusieurs types : de type 1 si la réaction à l'exception est limitée à une modification de la valeur des paramètres du schéma de commande de la RT ; de type 2 si la réaction requiert l'activation d'autres RTs ; de type 3 si l'exception est considérée comme "fatale" et dont la réaction entraîne l'arrêt du système.
- les postconditions, sont les signaux émis en fin d'exécution. Elles sont utilisées comme conditions à une terminaison normale de la tâche.

L'implantation d'une RT est faite suivant un ensemble de *Module-Tasks* (MTs), correspondant à des processus communiquants. Un MT sert à implanter :

- le schéma de commande périodique d'une RT.
- les différents *observers* en charge de la surveillance des signaux associés aux pré et post-conditions et exceptions.
- les comportements réactifs non-périodiques, responsable de l'activation/désactivation des tâches périodiques. Ces comportements sont activés par les signaux émis par les *observers*.

Une *Robot-Procedure* (RP) est une composition de RTs et/ou de RPs, ce qui permet une construction hiérarchique d'une architecture de contrôle. Une RP permet de définir une action robotique à un niveau d'abstraction quelconque, de la spécification d'une action simple (constituante d'une mission) jusqu'à la spécification d'une mission complète. Une RP permet de spécifier un arrangement logique et temporel de RTs et RPs permettant d'atteindre un objectif dans un contexte donné et de manière fiable, c'est-à-dire en exécutant des actions correctives dans le cas d'erreurs lors de l'exécution. Une RP est constituée de :

- un comportement logique défini, comme pour les RTs, par un ensemble de signaux d'entrée et de sortie (pré et post-conditions, exceptions).
- un *programme principal* représentant son exécution nominale. Il est composé de RTs, de RPs et d'opérateurs de composition permettant d'exprimer : l'exécution séquentielle, l'exécution parrallèle, des conditions, des itérations, des protocoles de rendez-vous, etc. Ce programme est chargé du séquencement des actions à effectuer et de la supervision de la réalisation de ces actions.
- un ensemble traitements de récupération associés aux exceptions de type 2 susceptibles d'être levées par les RTs et RPs le constituant.

L'organisation proposée dans ORCCAD est celle présentée dans la figure 3.5. S'il y a conceptuellement deux couches dans une architecture ORCCAD, la couche *application* peut être composée d'une hiérarchie de RPs. On peut donc considérer que la couche *application* peut être décomposée en un ensemble de sous-couches. Dans la couche *commande* (la plus basse) se trouvent les asser-

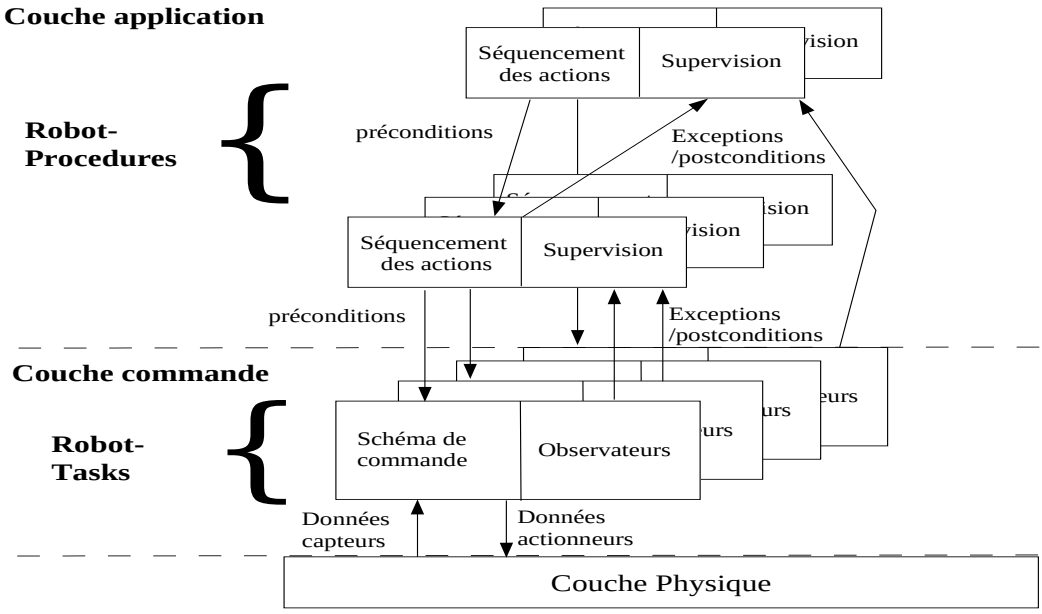


FIG. 3.5 – Modèle d'organisation d'architectures ORCCAD

vissements et observations. Dans la couche *application*, les RPs sont chargées du séquencement des actions, réalisées soit par activation de RTs, soit par activation de RPs. Dans cette approche, il est nécessaire de décrire pour chaque RP, l'ensemble des séquences de déclenchement de RTs ou RPs, amenant à la réalisation d'une action. Le modèle d'organisation implicite d'ORCCAD permet de concevoir des architectures mixtes. En effet, il est par exemple possible pour une RP située dans la couche la plus haute, de décider du lancement d'un asservissement ou d'une observation, situés dans la couche la plus basse. Ceci revient donc à permettre au transfert d'information (en particulier l'information de perception) de sauter des couches. La décision d'adaptation du contrôle peut donc se faire directement au niveau adéquat, sans que l'information à partir de laquelle cette adaptation est décidée, ait à traverser toutes les couches intermédiaires (dans ce cas, les sous-couches de la couche application). Ainsi, si ORCCAD demeure essentiellement basée sur une approche hiérarchique, elle permet néanmoins aux contrôleurs de gagner en réactivité via ce "saut" de couche lors du transfert d'information.

ORCCAD a déjà fait ses preuves à travers la réalisation de plusieurs projets robotiques. Elle semble à l'heure actuelle, une des approches les plus abouties en ce que concerne les capacités d'analyse (formelle, par simulation) d'un contrôleur logiciel. Ces possibilités d'analyse sont applicables indépendamment à chaque RT ou RP, ce qui permet, selon les auteurs, de créer des entités logicielles fiables. L'aspect fondamental de l'approche ORCCAD, est l'outillage logiciel proposé pour la supporter. Cet outillage permet l'analyse/vérification de propriétés logiques et temporelles des RTs et RPs [49]. La vérification des propriétés logiques se fait via génération du code ESTEREL équivalent aux RPs et RTs, puis par analyse de ce code via l'outil MAuto. L'analyse des propriétés temporelles se fait via la génération du code des RTs et RPs dans le langage Timed-Argos et via l'utilisation de l'outil Kronos. Il est intéressant de noter que les réseaux de Petri ont été utilisés pour spécifier et analyser le comportement des RTs [105]. D'autres outils permettent de générer le code exécutable (C++/ESTEREL) de leur application, ou encore de générer du code utilisable dans l'outil de simulation SIMPARC.

Le pouvoir d'expression des langages et outils de programmation d'ORCCAD est quelque peu limité en ce qui concerne la description et la gestion des missions. En effet, ORCCAD a une approche "programmatique" de la description des missions, ce qui implique que pour chaque mission ou action, l'utilisateur doit décrire toutes les façons dont elle peut se dérouler et ce, dans un langage dont la syntaxe est proche de celle d'un langage de programmation. Les entités en charge de la décision d'adaptation du contrôle (i.e. les RPs) sont intimement liées à une partie de la mission. Si ceci n'est pas une limitation pour les couches basses d'une architecture, cela semble être une limitation pour la conception d'entités en charge des couches décisionnelles, pour lesquelles le nombre de cas à gérer rend une action humainement impossible à décrire complètement. Ceci nous fait penser qu'ORCCAD reste une solution pour des problèmes de "bas niveau" dans une architecture de contrôle.

3.5 Chimera

Chimera [110] [109] est une méthodologie de développement d'architectures de contrôle, proposée par l'Université Cannergie Mellon de Pittsburg et par l'université du Maryland. Elle s'accompagne d'un système d'exploitation temps-réel spécifique [111] et d'outils de programmation.

Chimera repose sur la notion de *port-based object* pour ce qui concerne les problématiques de réutilisation et de composition. Les *port-based Objects* (PBO) sont des objets dédiés au développement de systèmes temps-réel. Les caractéristiques d'un PBO sont les suivantes :

- Un PBO possède toutes les caractéristiques d'un objet standard, c'est-à-dire un état interne, l'encapsulation du code et des données, et un ensemble d'opérations.
- Un PBO possède un ensemble de ports d'entrée, de ports de sortie et de ports de ressource, via lesquels il communique. Chaque port est associé à une variable que l'objet peut lire (port d'entrée) ou modifier (port de sortie). Les ports de ressources sont utilisés pour les communications extérieures au sous-système auquel appartient le PBO, par exemple avec l'environnement physique (capteurs, actionneurs), avec une interface utilisateur ou avec d'autres sous-systèmes (cf. plus loin). Les ports d'entrée et de sortie sont associés à des données qui peuvent être des constantes (servant lors de l'initialisation) ou des variables.
- Chaque PBO possède son propre automate, qui définit son comportement périodique lors de l'exécution. Il encapsule une tâche spécifique, qui s'exécute de façon autonome vis à vis des autres tâches d'une application. A chaque cycle, la tâche associée au PBO lit sur ses ports d'entrée les données les plus récentes, puis exécute un ensemble d'opérations prédéfinies, et enfin met à jour les valeurs de ses ports de sortie, modifiées par le cycle d'exécution.

Les PBOs sont des entités qui encapsulent des fonctionnalités de calcul exécutées de façon périodique (e.g. calcul de modèles cinématiques, loi de commande, générateurs de trajectoire, etc.). Ils sont composés via leurs ports d'entrée ou de sortie, en respectant la cohérence des noms de variables associés aux ports. La composition des PBOs résulte en un ensemble de tâches périodiques qui s'exécutent en concurrence et qui peuvent être déployées sur un ou plusieurs processeurs. Les PBO communiquent à travers leurs ports d'entrée et de sortie autour d'une mémoire partagée contenant les données les plus à jour pour chaque variable. Des services système sont présents dans l'OS Chimera, afin de gérer l'intégration et la composition des PBOs dans une architecture, par exemple des services de communication, de partage de tables de variables, d'ordonnancement ou de gestion des interfaces utilisateurs.

Les auteurs catégorisent les PBOs en trois grandes familles :

- Les PBOs *génériques*, qui ne sont ni dépendants d'une application donnée, ni d'un type de matériel spécifique.
- les PBOs *dépendant du matériel*, qui ne peuvent être exécutés que lorsque un matériel spécifique est utilisé dans une application robotique. Ces PBOs permettent de transformer les données

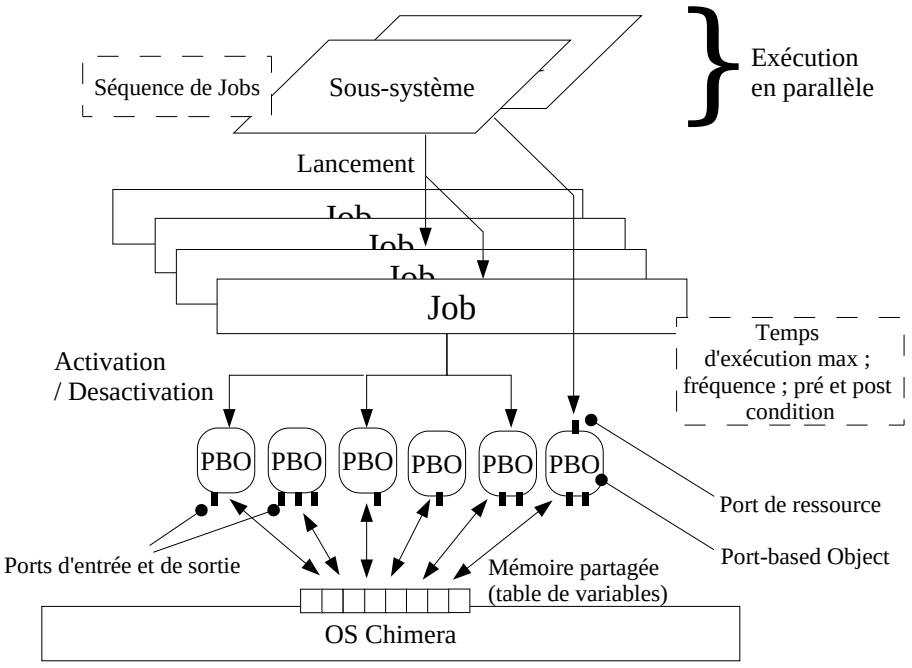


FIG. 3.6 – Modèle d'organisation d'architectures Chimera

spécifiques au matériel en données de plus haut niveau d'abstraction, ou permettent d'optimiser des PBOs génériques pour un matériel donné.
– Les PBOs *dépendants des applications*, qui sont utilisés pour implanter des détails spécifiques d'une application, et qui ne sont donc pas réutilisables à travers différentes applications.

Une composition de PBOs est encapsulée dans une entité appelée *Job*. Un *Job* représente une action de complexité variable ayant une finalité précise pour le contrôle du robot (e.g. "aller vers le point X", "chercher destination visuellement et déplacer vers cette destination", etc.). A chaque *Job* sont associées des propriétés temporelles paramétrables (e.g. fréquence d'exécution, temps d'exécution maximal, etc.) ainsi qu'une pré-condition et une post-condition. Un *sous-système de contrôle* contient une séquence de *Jobs*. Cette séquence décrit une exécution successive de *Jobs* du *sous-système de contrôle*, qui permet de réaliser une étape de la mission du robot. La transition entre deux *Jobs* se fait lorsque la post-condition du premier *Job* et la pré-condition du second *Job* sont toutes deux satisfaites. Cette transition se traduit sous la forme d'une reconfiguration des PBOs actifs : certains sont arrêtés, d'autres activés ou gardés en activité. Une application dans sa globalité est composée d'un ensemble de *sous-systèmes de contrôle* qui s'exécutent en parallèle et qui peuvent agir de façon coordonnée. Un *sous-système de contrôle* peut agir directement sur un PBO d'un autre sous-système en utilisant son port de ressource (par exemple pour reconfigurer certains paramètres d'exécution de ce PBO). C'est par le biais des ports de ressource des PBO que les *sous-systèmes de contrôle* peuvent se coordonner en influençant directement leur exécution dans leur couche basse (i.e. l'exécution des PBOs).

Chimera est une méthodologie de développement complète et outillée [108]. Chimera repose sur le modèle théorique dit des *port automatons* qui permet à des outils logiciels de se nourrir des informations temporelles afin d'analyser l'ordonnancement de l'exécution des PBO sur chaque processeur, ou leurs communications par mémoire partagée au sein d'un même *Job*. Enfin Chimera, repose sur un

environnement logiciel de développement spécifique [61], permettant de décrire les différentes couches d'une application de contrôle (PBO, Jobs, sous-systèmes de contrôle).

De la même façon qu'ORCCAD, Chimera propose un modèle d'organisation (implicite) qui permet d'identifier et de rendre réutilisables les entités des couches basses, qui encapsulent les asservissements et observateurs. On retrouve également la même volonté que dans CLARATy de mettre en relation de façon explicite le contrôle du robot, avec les éléments matériels contrôlés. Néanmoins, il semble difficile de concevoir des missions de grande complexité, en particulier celles où interviennent la planification et l'adaptation du contrôle, en fonction d'événements. En effet, le langage proposé pour décrire les séquences de *Jobs* ne semble pas posséder la puissance d'expression adéquate à la description de comportements de haut niveau décisionnel. Tout ceci rend complexe son utilisation directe dans un contexte de robotique de service.

L'originalité de ce modèle d'organisation des architectures de contrôle réside dans le fait qu'il permet de voir un contrôleur comme un ensemble de *sous-systèmes de contrôle*, ayant chacun une finalité précise dans la mission d'un robot (e.g. manipulation d'objets via un bras mécanique, déplacements d'un véhicule, plate-forme de téléopération, vision active, etc.). Un *sous-système de contrôle* peut influencer directement l'exécution de la couche basse d'un autre (ou plusieurs) *sous-système de contrôle*. Ce mécanisme reflète un des principes essentiels des architectures mixtes qui est de permettre le "saut" des couches lors du transfert de l'information (ici, de la couche la plus haute vers la couche la plus basse). Par exemple, le sous-système de vision active peut influencer le sous-système de contrôle des déplacements et inversement ; une plate-forme de téléopération peut influencer les sous-systèmes de manipulation ou de déplacement (e.g. consignes de position), etc. Dans ce cas, la réactivité du contrôleur résultant est forte, puisque l'influence des *sous-systèmes de contrôle* les uns sur les autres est directement prise en compte dans la couche la plus basse. Néanmoins, cette forme de coordination des sous-systèmes peut se révéler très complexe à exprimer et à contrôler, car il n'existe pas à notre connaissance de mécanismes permettant de gérer les cas conflictuels (e.g. un même PBO influencé par plusieurs sous-systèmes en même temps).

3.6 Architecture du LIRMM

Depuis 2001, le LIRMM étudie les méthodologies de conception d'architectures de contrôle, en tentant d'une part de répondre au problème de l'organisation au sein des architectures de contrôle mixtes [7], et d'autre part, en proposant des outils et méthodes permettant de modéliser le comportement des entités constitutives, à partir de réseaux de Petri de haut niveau [35].

L'architecture de contrôle (cf. fig. 3.7) initialement proposée par notre équipe [34] pour le contrôle d'un manipulateur mobile, est fondée sur un découpage d'un contrôleur logiciel en une hiérarchie de couches. Chaque couche est constituée d'un ensemble d'entités informatiques ayant des types prédéterminés. L'activité des entités situées dans une couche donnée est soumise à l'activité des entités des couches plus hautes. La hiérarchie des entités permet en particulier que l'adaptation soit réalisée dans la couche adéquate, sans faire appel, si possible, à des procédures de la plus haute couche décisionnelle et donc sans remettre en cause l'intention initiale provenant de cette couche. Dans toutes les couches, l'adaptation est dirigée par un ensemble d'événements [36]. Un événement est défini comme un phénomène d'origine proprioceptive ou extéroceptive, d'origine temporelle ou d'origine externe (i.e. provenant d'un opérateur ou d'un autre robot). Les événements sont détectés et émis par différentes entités logicielles.

De cette proposition émerge la notion centrale de *ressource robotique*, que nous appellerons plus simplement *ressource*. Une *ressource* est un sous-ensemble cohérent d'un robot, incluant un sous-ensemble de sa partie opérative et un sous-ensemble de son contrôleur, qui est dédiée exclusivement au contrôle de cette partie opérative. Un exemple de *ressource* est un manipulateur (bras mécanique,

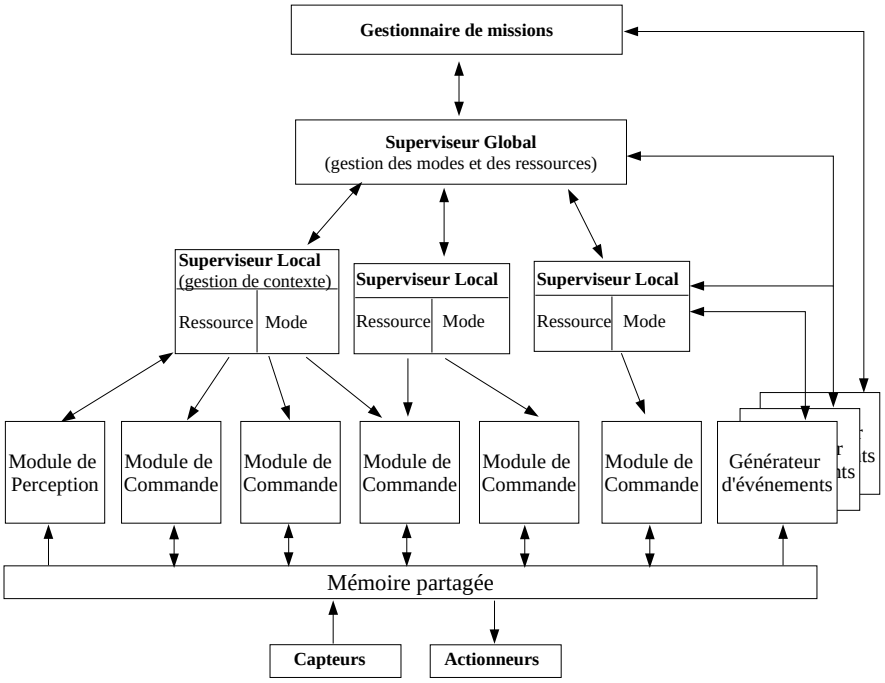


FIG. 3.7 – Modèle d'organisation des architectures du LIRMM

capteurs, actionneurs et logiciel du contrôle du bras mécanique) ou encore un mobile (véhicule, capteurs, actionneurs, etc.). Chaque *ressource* peut fonctionner dans un ensemble de *modes de fonctionnement* différents et exclusifs. La notion de *mode* permet en particulier de séparer les différentes options qu'à un utilisateur pour contrôler une *ressource* : autonomie, téléopération, coopération, etc. Deux modes ont été principalement envisagés : le mode autonome, correspondant au fait que la *ressource* reçoit d'un opérateur des directives de haut niveau d'abstraction (e.g. "aller à") ; le mode téléopéré correspondant au pilotage direct de la *ressource* par l'opérateur (e.g. suivi de consignes d'asservissement). Un robot est, in fine, constitué d'un ensemble de ressources pouvant être contrôlées simultanément dans des *modes* potentiellement différents (e.g. manipulateur autonome et mobile téléopéré pour un robot manipulateur mobile). Chaque *ressource* peut, en fonction de ses *modes de fonctionnement*, réaliser un ensemble d'actions (e.g. positionner le bras mécanique) afin de réaliser l'intention (de l'opérateur). Elle effectue une série d'actions prédéterminée, afin de réaliser une intention donnée. Chaque action correspond (potentiellement) à l'exécution synchronisée et séquentielle de plusieurs lois de commande.

L'architecture de contrôle du manipulateur mobile est constituée d'un ensemble de modules logiciels. Chaque module possède des traitements et des données propres, délègue certains traitements à des modules de plus basse couche dans l'architecture et émet des événements vers des modules de couches supérieures. Un module permet d'implanter une entité d'une couche quelconque, par exemple une boucle de perception ou une boucle d'asservissement situées dans la couche la plus basse. La figure 3.7 présente les différents types d'entités manipulés dans le modèle d'organisation sous-jacent à cette architecture. Le gestionnaire de mission est en charge des prises de décision de plus haut niveau dans l'architecture (e.g. planification) ; le superviseur global dirige l'utilisation des *ressources* dans différents *modes* en fonction d'ordres (provenant du gestionnaire de mission ou d'un opérateur) et du contexte (e.g. possibilité de panne matérielle ou logicielle d'une ressource). Chaque superviseur local est chargé du fonctionnement d'une *ressource* donnée dans un *mode* donné, en fonction d'ordres pro-

venant du superviseur global. Les générateurs d'événements sont dynamiquement configurés par les modules de niveaux supérieurs afin de détecter et de leur notifier les événements nécessaires à l'évaluation du contexte. Chaque module de commande est chargé de l'application d'une loi de commande spécifique.

Cette proposition est intéressante car elle propose de modéliser le comportement asynchrone et parallèle des modules, par le biais des réseaux de Petri à Objets [35]. De la même façon que les proposition du LAAS et qu'ORCCAD, l'aspect "validation" y est donc central. Contrairement à d'autres propositions (celles du LAAS et ORCCAD par exemple), celle du LIRMM n'est sous-tendue par aucun environnement et outil de développement logiciel permettant d'automatiser certaines phases du développement (e.g. analyse, génération de code).

A l'instar des autres approches présentées, le modèle d'organisation des architectures de contrôle du LIRMM identifie clairement les fonctionnalités associées à chaque niveau. Ce modèle d'organisation d'architecture est considéré comme mixte, puisque le transfert d'information peut "sauter" des couches, afin que l'information soit traitée directement au niveau adéquat. En particulier, l'information issue de générateurs d'événements (i.e. des événements) peut être propagée directement vers différentes couches supérieures d'une architecture, en fonction du ou des modules où cette information est utilisée pour déclencher une réaction.

Le modèle d'organisation des architectures proposé par le LIRMM identifie des types d'entités (modules de commande, modules de perception, superviseurs locaux, etc.) à grain plus fin et plus spécialisés que dans d'autres modèles. C'est, selon nous, un pas vers une meilleure compréhension et une meilleure maîtrise de la construction d'une architecture de contrôle, qui permet d'optimiser sa modularité. Cependant, ce modèle d'organisation possède des limitations d'un point de vue de l'ingénierie logicielle. Elle ne met pas en correspondance les concepts "métiers" manipulés et l'architecture elle-même. Premièrement, différents modules font référence à des éléments contrôlés communs (e.g. éléments de la partie opérative) sans que cela puisse être exprimé clairement dans l'architecture (à la différence de ce que propose CLARATy). Par exemple, un superviseur local d'un bras mécanique et un module de commande ne peuvent interopérer que si le module de commande définit un asservissement sur le même type de bras. Puisque ce genre de relation n'existe pas dans le modèle actuel, la réutilisation s'avère difficile à appréhender. Deuxièmement, les données et traitements utilisés ne sont pas clairement identifiés et catégorisés. Par exemple, deux modules de commande pour un même type de bras peuvent utiliser un même modèle géométrique direct ou un même générateur de trajectoire sans que cette relation puisse être exprimée clairement. La catégorisation de ces traitements, par exemple sous la forme d'objets représentant l'élément matériel considéré, permettrait de mieux réutiliser les traitements relatifs à la commande de ces éléments. Enfin, les modules manipulent et/ou interagissent avec des entités abstraites non explicitement représentées dans une architecture. Ces entités sont les *ressources*, les *modes* et les actions. Par exemple, le superviseur global manipule des *ressources* et des *modes* mais ces entités ne sont pas représentées dans le modèle d'organisation, et ne sont donc pas considérées commes des entités réutilisables (ou au moins identifiables de façon indépendante). Nous pensons que, pour renforcer la modularité et favoriser la réutilisation, le modèle d'organisation des architectures de contrôle doit refléter aussi loin que possible, les concepts manipulés par les architectes.

3.7 AURA

AURA (AUtonomous Robot Architecture) est une architecture générique et extensible, dédiée aux problèmes de navigation et de mobilité. Elle est développée depuis le milieu des années 80 [10] par l'université du Massachussetts. Elle a évolué et a été progressivement adaptée à de nombreux

domaines d'application [11] (sous-marin, manipulateur mobile terrestre, militaire, flotille de robots, etc.).

Le modèle d'organisation sous-jacent à l'architecture AURA (cf. fig. 3.8) définit deux couches : une couche *décisionnelle* et une couche *réactive*. La couche *décisionnelle* est en charge de toutes les décisions de haut niveau d'abstraction. Elle est décomposée en une hiérarchie de trois sous-couches : un planificateur de mission (*mission planner*), un navigateur (*spatial reasoner*) et un séquenceur de plans (*plan sequencer*). Le planificateur de mission est situé au plus haut de cette hiérarchie, il a pour rôle d'établir les buts de haut niveau du robot et les contraintes qu'il doit respecter. Il sert principalement, à notre connaissance, d'interface avec les opérateurs humains qui définissent les missions. Le planificateur de mission définit les objectifs du robot sur un horizon "à long terme", à partir des informations données par un opérateur et de la mémoire "à long terme" du robot (carte de son environnement). Le navigateur utilise et met à jour la connaissance cartographique de l'environnement du robot, stockée dans la mémoire "à long terme" (*representation*), afin d'en déduire les chemins que le robot peut ou doit emprunter pour réaliser les objectifs (définis précédemment). Le séquenceur de plan calcule, en fonction de ces chemins, le plan d'action permettant de réaliser sa mission. Ce plan est défini comme un diagramme d'états, où chaque état représente une action que le robot doit réaliser et où chaque transition entre états représente le changement d'action. Chaque transition est associée à un ensemble de conditions qui correspondent aux différents stimulis qui vont engager le changement d'action (e.g. objectif atteint, impossibilité d'atteindre un but, etc.). Chaque état d'un plan d'action correspond à un ensemble de comportements réactifs et un ensemble de schémas de perception en activité (cf. plus loin) et en interaction. Une fois un plan d'action défini par le séquenceur de plans, ce dernier transfère à la couche *réactive* la responsabilité d'exécuter le plan.

La couche *réactive* est constituée d'un contrôleur de schémas (cf. *Schema Controller*). Le contrôleur de schémas se charge d'instancier, de composer et d'exécuter plusieurs schémas en fonction du plan d'action. Les schémas sont des entités asynchrones qui sont soit des schémas de perception, soit des schémas moteurs. Les schémas de perception produisent, à partir des données capteurs, les stimulis nécessaires à l'exécution des schémas moteurs. Ils produisent également des stimulis, utilisés directement par le contrôleur de schémas pour engager le changement d'action (i.e. le changement d'état décrit au sein du plan). Enfin ils peuvent générer des données utilisées pour enrichir la mémoire à "long terme" du robot. Les schémas moteurs définissent des comportements réactifs à finalité variable (e.g. évitement d'obstacles, déplacement vers un point, éloignement d'un point, etc.). En fonction des stimulis reçus (donc en fonction de leur composition avec des schémas de perception, définie pour chaque état du plan), ils génèrent des vecteurs de commande qui sont transmis au contrôleur de schémas. Ce dernier se charge de l'arbitrage, qui consiste à sommer et pondérer les vecteurs de commande générés par l'ensemble des schémas moteurs concurrents. La pondération est basée sur une "valeur de gain" associée à chaque schéma moteur, en fonction de leur importance relative pour la réalisation d'une action (i.e. la valeur de gain de chaque schéma moteur est définie dynamiquement en fonction de l'état du plan d'action). Par exemple, la valeur de gain du comportement permettant d'éviter un obstacle est plus forte que celle d'un comportement permettant de se déplacer vers un point, afin d'assurer l'intégrité physique du robot. L'arbitrage permet de générer l'unique vecteur de commande que le contrôleur de schémas applique périodiquement aux actionneurs.

Une fois que la couche *réactive* démarre son exécution, la couche *délibérative* n'est pas réactivée jusqu'à ce qu'une faute (e.g. le robot n'arrive plus à se déplacer) soit détectée par le contrôleur de schémas. A ce moment, l'exécution du niveau décisionnel reprend de façon ascendante. En premier lieu, le séquenceur de plan essaie, à partir des rapports d'exécution du contrôleur de schémas (contenant la mémoire "à court terme" du robot) de modifier le plan afin de sortir le robot de sa situation. S'il n'arrive pas à établir un plan satisfaisant, le navigateur est activé. Ce dernier se base sur les informations contenues dans sa mémoire à "long terme" (i.e. sur l'environnement global qu'il a déjà exploré) afin de générer un nouveau chemin global. Si cela s'avère inefficace, le planificateur

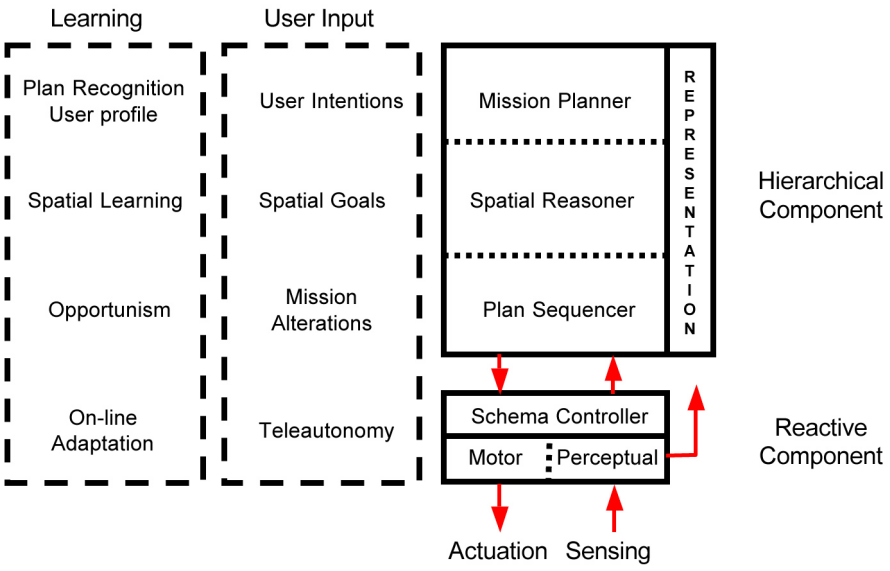


FIG. 3.8 – Modèle d'organisation d'architectures AURA [11]

de mission est activé. Celui-ci se charge alors d'avertir l'opérateur de son impossibilité à trouver une solution (avec éventuellement les données permettant à l'opérateur de comprendre cet échec). Il attend alors que l'opérateur reformule la mission ou demande son abandon.

AURA reste une proposition essentiellement centrée autour des problèmes de navigation et de mobilité des robots. La couche *décisionnelle* (en particulier les niveaux planificateur de mission et navigateur) se concentre essentiellement sur la problématique de navigation dans un environnement potentiellement évolutif. AURA a également été utilisé pour concevoir des architectures de contrôle abordant des problématiques plus larges que celle de navigation. Mais dans ce cas, le navigateur et le planificateur de mission n'ont pas été, à notre connaissance, intégrés.

L'avantage de la proposition AURA est qu'elle repose sur un modèle d'organisation relativement neutre par rapport aux outils logiciels et algorithmes utilisés dans les différentes couches. Ceci a permis le test et l'intégration progressive de nouveaux algorithmes plus efficaces ou plus génériques. De notre point de vue, ce modèle d'organisation est celui qui intègre le mieux les concepts sous-jacents aux architectures mixtes : le modèle d'organisation sous-jacent aux architectures comportementales y est explicitement intégré avec celui des architectures délibératives (approche hiérarchisée). Une architecture logicielle est conçue, dans la couche *décisionnelle*, suivant une organisation hiérarchique, dans laquelle des mécanismes classiques de l'intelligence artificielle et de la robotique sont utilisés (navigation, planification, ordonnancement d'action). Dans la couche *réactive*, une architecture logicielle est conçue à partir de la composition non complètement maîtrisée (dûe à l'utilisation d'un mécanisme d'arbitrage) de comportements réactifs, qui tend à faire émerger un comportement global du robot.

3.8 Reflexion sur les architectures des contrôleurs

Cette étude des modèles d'organisation des architectures logicielles de contrôle a permis de faire émerger certains concepts et principes qui nous semblent majeurs. Nous dressons en premier lieu un constat global sur l'organisation de ces architectures, puis nous présentons les concepts et principes qui nous semblent les plus importants.

3.8.1 Constat Général

Toutes les approches présentées sont qualifiées de mixtes ou hybrides dans la littérature, mais elles sont néanmoins différentes. Nous présentons par la suite les points qui nous semblent caractériser ces différentes approches :

- Certaines approches comme celles du LIRMM [36] et ORCCAD [27] proposent une organisation en couches, mais pour gagner en réactivité, le transfert d'information (données d'états ou données représentant les décisions) peut sauter des couches. Par exemple, un événement détecté dans une couche basse peut-être transmis directement à la couche supérieure concernée par la réaction à cet événement, en sautant les couches intermédiaires et inversément pour le transfert d'une décision d'une couche haute vers une couche basse. Les avantages de ces architectures sont les mêmes que les architectures hiérarchisées avec, en plus, un gain en réactivité. Néanmoins, la possibilité de sauter des couches pendant les transferts d'information complique la gestion des interactions.
- Certaines architectures, notamment les architectures AURA [11], couplent une approche hiérarchisée avec une approche comportementale. La couche la plus basse est constituée de comportement réactifs et d'un mécanisme d'arbitrage ainsi que d'observateurs. La couche supérieure est chargée de contrôler l'exécution des comportements réactifs et observateurs (activation/désactivation) et de configurer le mécanisme d'arbitrage en fonction de la tâche que doit accomplir le robot et de sa localisation dans l'environnement. Ces architectures ont l'avantage d'être globalement maîtrisables par l'homme, tout en reposant sur un comportement global émergent. Evidemment, la réalisation de la couche basse reste aussi compliquée que dans les architectures comportementales (complexité du mécanisme d'arbitrage et complexité des interactions).
- Certaines architectures proposent une forme d'organisation que nous qualifions de *systémique* : les entités logicielles sont regroupées en "blocs" qui agrègent le contrôle d'une partie identifiée de la partie opérative d'un robot. C'est, par exemple, le cas des *sous-systèmes de contrôle* de Chimera [109] : un sous-système de vision encapsule le contrôle des caméras embarquées et les traitements réalisés sur les flots d'images ; un sous-système de locomotion contrôle le véhicule qui permet au robot de se déplacer dans l'environnement (e.g. asservissement, planification et supervision des mouvements). Le contrôleur est, in fine, constitué d'un ensemble de sous-systèmes en collaboration. Un concept proche existe implicitement dans le modèle d'organisation du LIRMM, il s'agit des *ressources*. Enfin, cette même vision par agrégation, existe également dans IDEA [87]. Un agent IDEA encapsule différentes entités (planificateur, planificateur réactif, supervision, commande) qui sont utilisées pour contrôler une partie identifiée de la partie opérative d'un robot. Cette approche rend la réutilisation d'une sous-architecture de contrôle plus intuitive que dans les autres approches, car elle couple explicitement l'organisation d'une architecture de contrôle avec le système physique contrôlé.
- Les architectures CLARATy [88] ont pour originalité de réifier les connaissances que le robot possède sur le "monde" sous forme de classes (e.g. données représentant l'environnement et leurs traitement associés ; propriétés physiques, géométriques et cinématiques de la partie matérielle du robot). Cela permet de séparer la description du "monde", de la description des activités que le robot a pendant sa mission (la façon dont il prend et applique des décisions à partir de ces connaissances) qui est plus spécifiquement liée à sa finalité. L'avantage majeur de cette approche est de favoriser la réutilisation des entités de représentation du "monde réel", mais la partie "décisionnelle" qui est décrite de façon monolithique, reste non-réutilisable.

3.8.2 Critères d'organisation en couches

Globalement, les architectures de contrôle mixtes sont organisées en *couches*, i.e. organisées de façon hiérarchique. Il est bon de noter que les couches sont avant tout des "vues de l'esprit" qui

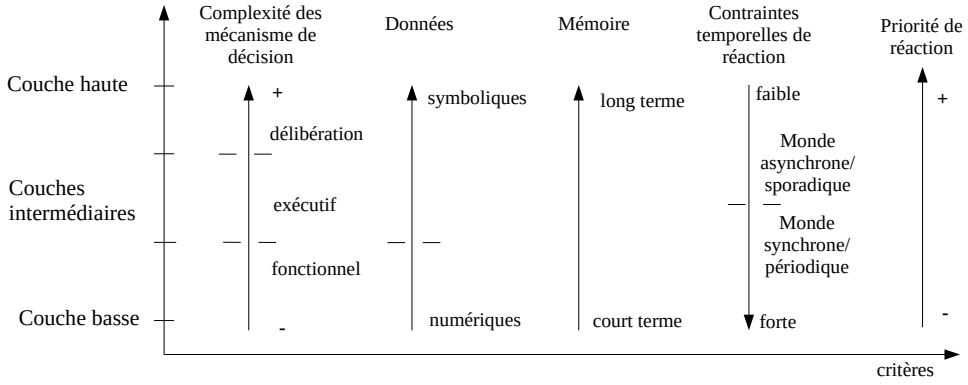


FIG. 3.9 – Critères d'organisation en couches

n'existent pas nécessairement de façon explicite au niveau logiciel. Une organisation en couches sert donc essentiellement comme un guide pendant la conception d'une architecture de contrôle. Elle définit un ensemble de critères (cf. fig. 3.9) qui aident les développeurs à structurer leurs modèles.

Le premier critère est celui associé aux "niveaux de décision". Chaque couche est en charge d'un "niveau de décision" au sein d'un contrôleur. Plus la couche est basse dans l'architecture, plus les mécanismes de délibération et les données manipulées sont de "bas niveau". Plus la couche est haute dans l'architecture et plus les mécanismes de délibération sont complexes et les données abstraites. Par exemple, la couche fonctionnelle peut être considérée comme celle de plus bas "niveau de décision", puisqu'elle contient les lois de commande (qui sont considérées comme des comportements réactifs d'un robot), à partir desquelles sont réalisés les comportements plus complexes. Les données manipulées y sont essentiellement numériques. La couche décisionnelle est de plus haut niveau puisqu'elle contient les mécanismes de délibération les plus complexes, comme la planification et la supervision de missions. Les données manipulées dans cette couche sont essentiellement symboliques et représentent par exemple des cartes de l'environnement, des ordres, des événements. Le nombre de couches intermédiaires, quant à lui, peut varier en fonction du modèle d'organisation proposé. Plus la couche est haute et plus les données manipulées sont symboliques et abstraites. Ceci s'explique par le fait que les mécanismes décisionnels de haut niveau se basent essentiellement sur des algorithmes utilisant des données symboliques.

Une autre critère est celui de la gestion de la mémoire. Les propositions du LAAS, du LIRMM et AURA suggèrent, explicitement ou non, que la gestion de la mémoire au sein d'une architecture de contrôle est fonction de la couche. Plus une couche est haute dans l'architecture, plus la mémoire associée aux entités de cette couche est vue comme étant "à long terme". Dans les couches les plus hautes, la mémoire reflète par exemple l'environnement perçu par le robot lors de sa mission, ou le déroulement de la mission elle-même. Dans les couches les plus basses, la mémoire reflète par exemple les données capteurs lues ou les données actionneurs générées. Dans ce cas, la mémoire est très "volatile", l'historique des valeurs étant très restreint.

L'organisation des couches peut également être décidée en fonction des contraintes temporelles de réaction qui leur sont associées, ce qui constitue un autre critère. Plus une couche est basse dans l'architecture et plus l'exécution des entités qu'elle contient est temporellement contrainte. L'échelle de temps de réaction dans les couches les plus basses est faible (e.g. de l'ordre de quelques millisecondes). Plus une couche est haute et plus cette échelle de temps de réaction augmente (e.g. secondes, minutes, etc.). Cette échelle de temps de réaction permet de considérer que les entités

contenues dans les couches les plus basses s'exécutent dans un monde "synchrone" (i.e. périodique). Les entités contenues dans les couches les plus hautes s'exécutent dans un monde purement asynchrone (i.e. sporadique). Les entités contenues dans les couches intermédiaires s'exécutent soit dans un monde synchrone, soit dans un monde asynchrone, soit à la frontière entre ces deux mondes (une partie de leur comportement est périodique, l'autre partie est sporadique).

Enfin, une organisation d'architecture de contrôle peut être caractérisée par la priorité de réaction associée à chaque couche. Les entités contenues dans les couches les plus hautes doivent s'exécuter plus prioritairement que celles contenues dans les couches basses. Ceci permet d'assurer que les décisions prises à un "niveau de décision" soient répercutées prioritairement à celles des couches inférieures. Ce critère semble essentiel à la bonne gestion de l'adaptation d'un robot à son contexte (e.g. panne matérielle, obstacle dans l'environnement). Le changement de contexte peut, en effet, amener différentes couches à réagir. La réaction décidée par une couche supérieure doit être, a priori, prioritaire par rapport à celle décidée par une couche inférieure. En effet, la couche supérieure possède une vision "plus globale" du contexte du robot et prend donc des décisions plus "stratégiques".

L'ensemble de ces critères, résumés dans la figure 3.9, constituent de bons repères à partir desquels organiser une architecture de contrôle. Ils peuvent être utilisés comme base pour la définition de nouveaux modèles d'organisation d'architectures. Néanmoins, les couches n'y apparaissent que comme des abstractions d'un "niveau de décision" au sein d'un contrôleur, mais aucun détail sur ces couches n'est donné. En effet, une couche est faite d'un ensemble d'entités logicelles, comme c'est le cas dans la plupart des modèles présentés.

3.8.3 Entités logicielles

Un constat sur les architectures de contrôle, est que leur organisation en couches reflète non pas le robot lui-même, mais son système de décision/contrôle. Les entités de contrôle constituant les couches sont donc vues commes différentes parties de ce système. On peut établir un parallèle avec la manière dont on décompose les différentes parties du système nerveux central humain en différentes zones ayant des fonctions spécifiques (langage, émotions, vision, etc.). D'un autre coté, il est nécessaire de représenter les connaissances manipulées par les différentes parties du système de contrôle. Par analogie avec l'homme, on peut considérer que la zone du cerveau responsable de la vision possède une représentation interne de l'environnement physique à partir de laquelle elle peut interpréter les images.

C'est pourquoi nous distinguons deux sortes d'entités logicielles de natures différentes : les *entités de contrôle* et les *entités de représentation des connaissances*. Les *entités de contrôle* gèrent les activités liées à la prise de décision au sein du contrôleur. Les *entités de représentation des connaissances* permettent de réifier les connaissances que possède le contrôleur sur le "monde réel".

Entités de contrôle

Chaque couche d'une architecture de contrôleur est constituée d'un ensemble d'entités de contrôle en charge de la réalisation d'un ensemble de traitements. La diversité de ces entités est importante, allant d'un asservissement à des mécanismes de planification et de supervision de mission. D'un point de vue très abstrait, une entité logicielle est constituée d'un cycle *perception* - *décision* - *action* (cf. figure 3.10).

La *perception* (au sens commun) est le mécanisme à partir duquel une entité reçoit des données et définit son contexte courant. Ce contexte représente l'état de son environnement, dépendant du niveau d'abstraction auquel se situe l'entité (e.g. mesures capteurs, données reconstruites par observation, événements, carte de l'environnement, etc.) ainsi que son état propre (représentant sa propre activité). La *décision* (au sens large) repose sur un mécanisme via lequel une entité détermine le comportement

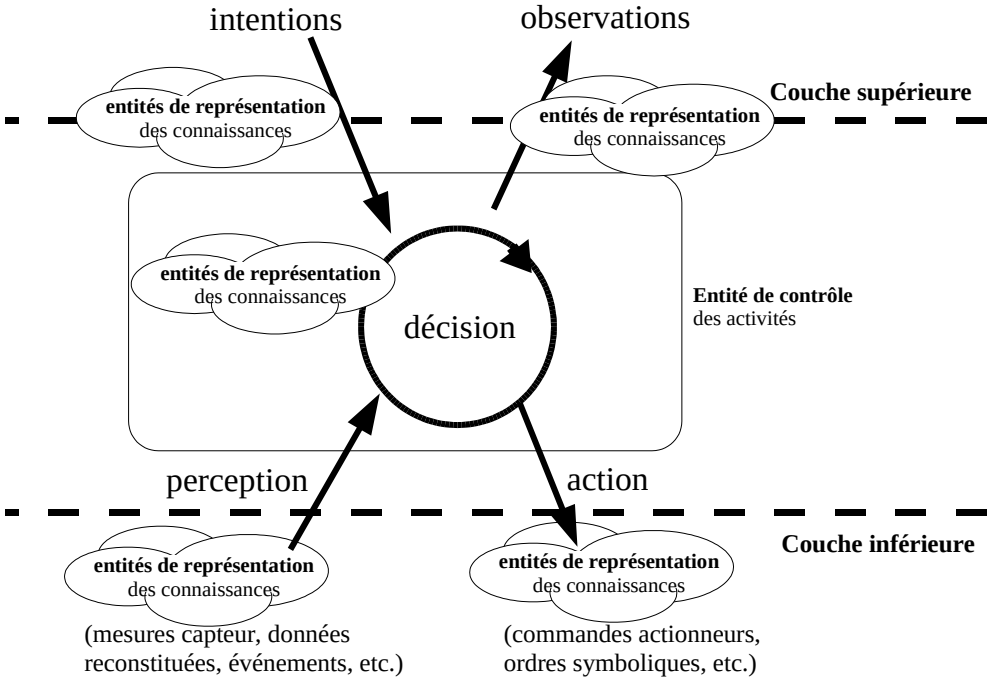


FIG. 3.10 – Schéma d'une entité logicielle de contrôle

qu'elle doit adopter en réaction au changement de son contexte. Par exemple, pour des entités de la couche la plus basse, le mécanisme de décision est vu comme le calcul périodique de la correction à apporter aux mouvements d'un robot, en fonction d'une loi de commande. Dans les couches les plus hautes, la réaction calculée peut être le résultat de l'exécution de mécanismes de planification ou d'apprentissage. L'*action* est le mécanisme via lequel une entité de contrôle influence l'activité des entités de contrôle des couches inférieures, afin qu'elles réalisent ses décisions. Par exemple, pour une entité de la couche la plus basse, l'*action* peut simplement consister à fixer la valeur des commandes envoyées aux actionneurs. Pour une entité d'une couche plus haute, l'*action* peut passer par de nombreuses interactions avec les entités des couches inférieures dans le but, par exemple, de reconfigurer/arrêter/démarrer leur exécution.

Ce cycle *perception* - *décision* - *action* d'une entité doit pouvoir être contrôlé, ou au moins observé. Sans moyen de contrôle, toutes les entités appartiendraient conceptuellement à la même couche et le modèle d'organisation sous-jacent serait alors purement comportemental. Une entité de contrôle doit donc être capable d'interagir avec des entités des couches supérieures, afin que celles-ci contrôlent son exécution. Pour cela, elle reçoit des *intentions* des couches supérieures, qui contrôlent ce cycle (activation/désactivation, paramétrage, demandes d'observation, etc.) et elle émet des *observations* (calculées pendant ce cycle) vers les couches supérieures. Le mécanisme de *perception* d'une entité de contrôle, se nourrit des *observations* émises par des entités de contrôle des couches inférieures, afin de connaître les états "observés" de l'environnement et de la partie matérielle, ou l'état des activités des couches inférieures. Le mécanisme d'*action* d'une entité de contrôle génère un ensemble d'*intentions* qui seront reçues par des entités de contrôle des couches inférieures.

Ce modèle d'entité logicielle de contrôle est totalement neutre vis à vis des mécanismes utilisés pour faire interagir des entités, ou mettre en œuvre leurs mécanismes de *décision*. Il est également suffisamment neutre pour généraliser les entités mettant en œuvre les mécanismes d'adaptation, les

asservissements ou les boucles d'observation (dans ce dernier cas l'entité n'a pas nécessairement de mécanisme d'*action*).

Entités de représentation des connaissances

Pour interagir, les entités de contrôle ont besoin d'échanger des données de niveaux d'abstraction quelconques. Ces données représentent les *connaissances* (cf. fig. 3.10) partagées par les entités de contrôle en interaction. La nature des connaissances partagées par une entité de contrôle varie en fonction de la couche avec laquelle elle interagit. Généralement, on peut considérer que les connaissances partagées avec une entité de couche supérieure sont du même ou d'un plus haut niveau d'abstraction (e.g. données symboliques) que celles partagées avec une entité de couche inférieure (e.g. données numériques). En interne, une entité de contrôle peut manipuler (dans son mécanisme de *décision*) des données qui représentent les *connaissances* qu'elle possède sur son contexte local et qu'elle ne partage avec aucune entité à l'exécution.

L'identification et la réification des *connaissances* utilisées au sein des différentes entités logicielles est, selon nous, essentiel. En effet, la réutilisation efficace des connaissances est d'autant plus importante que les connaissances manipulées sont complexes. D'autre part, ces connaissances sont à la base de l'interaction des entités logicielles, elles vont donc grandement conditionner leur capacités d'interopération. Enfin, ces connaissances sont manipulées en interne par chaque entité, elles sont à la base de la définition de leur comportement *perception - décision - action*. Par exemple, les connaissances relatives aux éléments matériels d'un robot (par exemple leur modèle géométrique) peuvent potentiellement être utilisées dans différentes couches ou au moins différentes entités de contrôle d'une même couche. Cette idée de représenter explicitement les connaissances peut être retrouvée, plus ou moins explicitement et sous différentes formes, dans les modèles d'organisation présentés. Dans CLARATy par exemple, les objets de la couche fonctionnelle représentent les éléments matériels constituant le robot ou certaines de ses stratégies de contrôle. Autrement dit, elles représentent les connaissances que la couche décisionnelle possède sur le robot, à partir desquelles elle délibère et contrôle la partie opérative du robot. Cette idée émerge de façon complètement différente dans l'approche du LIRMM, dans laquelle les objets véhiculés dans les RdPO représentent des connaissances locales à chaque entité (connaissance sur le robot physique, sur sa mission).

Il est possible de définir une *entité de représentation de connaissance* comme une entité logicielle, encapsulant données et traitements et permettant de représenter : une partie du monde sur lequel le contrôleur agit (e.g. carte de l'environnement, chemin dans l'environnement, partie opérative, données capteurs, etc.) ou une stratégie d'action et/ou d'observation de phénomènes sur ce monde (e.g. loi de commande, estimateur, générateur de chemin, etc.). Les *entités de contrôle* exploitent ces *entités de représentation des connaissances* afin de donner au robot le comportement global souhaité.

3.8.4 Synthèse d'un modèle générique

Les principes d'organisation cités nous amènent à faire une première ébauche des caractéristiques principales des architectures des contrôleurs. La figure 3.11 présente un modèle d'organisation d'architectures mixtes, issu de la synthèse de ces principes. Ce modèle met en évidence deux formes d'organisation orthogonales : l'organisation en couches et l'organisation systémique.

Organisation en couches

Ce modèle décompose conceptuellement une architecture de contrôle en un ensemble de couches superposées. Ces couches sont hiérarchisées en fonction des critères définis dans la première partie de cette section, la couche la plus basse étant située directement au dessus de la couche physique (abstraction du matériel utilisé pour contrôler le robot). Chaque couche est constituée d'un ensemble

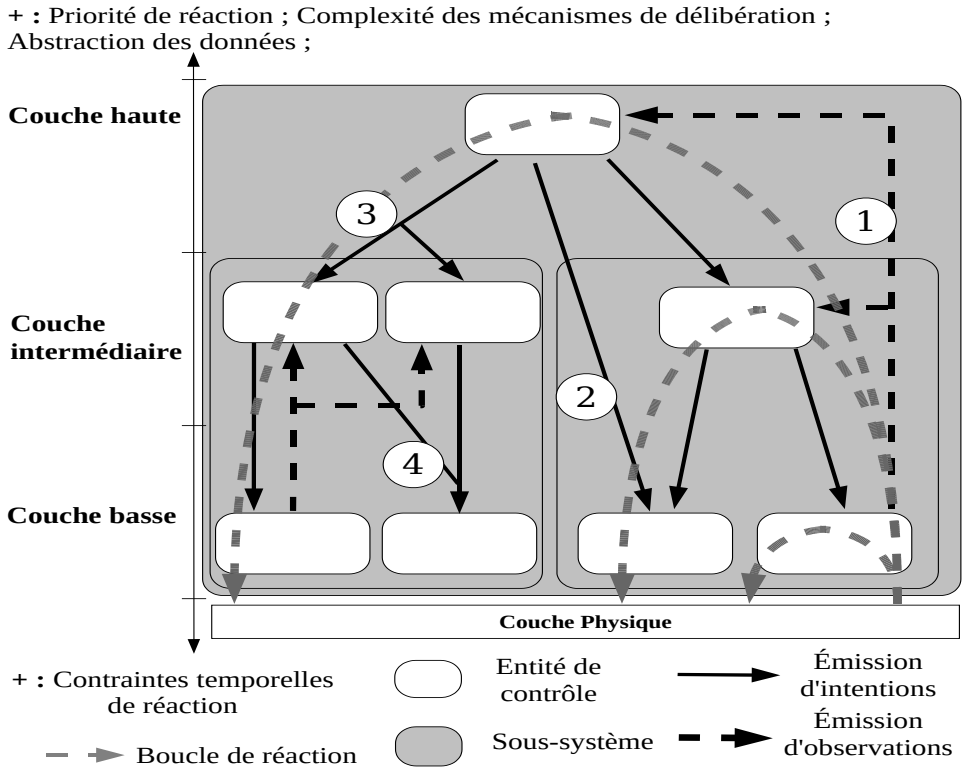


FIG. 3.11 – Modèle d'organisation général des architectures de contrôle mixtes

d'entités de contrôle qui relèvent d'un même "niveau de décision"(e.g. planification, asservissement, observation, échantillonnage des E/S, etc.). Le nombre de couches est laissé indéterminé dans un souci de généricité, chaque architecte pouvant décider du nombre et de la nature exacte des "niveaux de décision" nécessaires à la réalisation d'un contrôleur de robot.

Une caractéristique essentielle des architectures de contrôle mixtes réside dans la possibilité, pour toute couche, d'interagir avec des couches qui ne lui sont pas directement supérieures ou inférieures. Cette caractéristique permet au système de gagner en réactivité, en évitant que l'information ne soit traitée par toutes les couches intermédiaires.

Premièrement, l'information doit remonter directement de la couche où l'information est générée vers la couche capable de prendre la décision de réaction. Sur le modèle (cas 1, fig. 3.11), une observation peut être émise vers une entité de contrôle appartenant à une couche non-directement supérieure. Elle peut, de surcroit, être émise simultanément à plusieurs entités de contrôle qui appartiennent soit à une, soit à différentes couches supérieures. Par exemple, si une entité de contrôle de la couche la plus basse est capable de détecter un obstacle, elle peut notifier deux entités de contrôle qui vont décider de la réaction à appliquer en fonction de leur "niveau de décision", l'une reconfigurant les asservissements actifs (pour éviter de percuter l'obstacle), alors que l'autre réalise une nouvelle planification (pour prendre en compte la présence de l'obstacle dans les déplacements futurs du robot). Ainsi, à partir d'une même observation, plusieurs décisions peuvent être réalisées, d'une décision "rapide" dans les couches basses, vers une décision "longue" dans les couches hautes.

Deuxièmement, une fois la décision de réaction prise par une entité de contrôle, cette décision est propagée à travers les couches inférieures. Là encore, l'information doit pouvoir, si nécessaire, redescendre directement vers la couche où la réaction sera effectivement appliquée, sans traverser les couches intermédiaires. Sur le modèle (cas 2, fig. 3.11), une intention peut être émise vers une entité de contrôle d'une couche non directement inférieure. C'est par exemple une solution existante lorsqu'une téléopération directe est établie : la plate-forme opérateur (entité de contrôle de plus haut niveau décisionnel) envoie des consignes qui sont directement prises en compte par les asservissements, sans passer par des entités responsables de la planification de mouvements. Notons néanmoins que ceci pose des problèmes pour garantir la cohérence des contextes d'exécution locaux aux entités des couches intermédiaires et demande donc beaucoup de précautions dans sa mise en œuvre.

Cette organisation permet de mettre en place des boucles de réaction de manière plus "efficace" qu'avec une organisation hiérarchisée classique. Une boucle de réaction symbolise la transmission des informations à la base d'une adaptation du robot à son contexte d'exécution (défini par l'état de sa mission, de son environnement et de sa partie opérative). Une boucle de réaction dans la couche la plus basse correspond, par exemple, à un asservissement. Elle peut être directement mise en œuvre par une unique entité de contrôle. Une boucle de réaction peut faire intervenir plusieurs entités de contrôle : dans ce cas l'information (par exemple un événement) générée par une couche basse, remonte vers la couche supérieure capable de prendre la décision de réaction, puis redescend vers la couche basse capable d'appliquer la réaction à la couche matérielle. C'est sur ce dernier point que le modèle d'organisation mixte proposé est plus efficace, car ces boucles de réaction peuvent être réalisées sans traverser toutes les couches intermédiaires, comme c'est le cas dans les architectures délibératives (cf. fig. 3.1).

Organisation systémique

Le modèle proposé recommande une description systémique d'une architecture de contrôle (cf. fig. 3.11). Une architecture peut-être décomposée en un ensemble de systèmes, qui représentent chacun le contrôle d'une partie identifiée de la partie opérative du robot. Par exemple le système de manipulation agrège la (sous-)architecture de contrôle du bras mécanique. Nous généralisons l'approche en permettant à un (sous-)système d'être agrégé dans un autre système qui représente le

contrôle d'une partie matérielle plus importante. Par exemple, les sous-systèmes de locomotion et de manipulation devraient pouvoir être agrégés dans un sous-système contrôlant un robot manipulateur mobile dans sa globalité. Le modèle n'impose conceptuellement aucune limite au nombre de niveaux d'agrégation possible.

Un système contient un ensemble d'entités de contrôle relatives aux mêmes éléments matériels, et qui peuvent être réparties dans différentes couches. L'organisation systémique est donc orthogonale et complémentaire à une organisation en couches. Elle permet aux développeurs de s'abstraire de certains détails, en manipulant des entités d'abstraction supérieure.

Intégration de l'approche comportementale

Intégrer une approche comportementale à une architecture repose avant tout sur l'organisation des interactions entre entités de contrôle situées dans différentes couches. Cela semble uniquement concevable, à l'heure actuelle, dans les couches les plus basses d'une architecture de contrôle (au niveau des asservissements et contrôle des actionneurs). Cette approche est traduite sur le modèle en deux relations spécifiques. La première relation (cas 3, fig. 3.11) représente l'émission simultanée de deux intentions vers deux entités de contrôle en charge des asservissements. Par exemple, afin de réaliser un déplacement du véhicule du robot mobile, une demande de démarrage est envoyée à deux asservissements concurrents : un est en charge de suivre une trajectoire, et l'autre est en charge d'éviter des obstacles. La seconde relation (cas 4, fig. 3.11) représente le fait qu'une entité de la couche la plus basse, en charge du mécanisme d'arbitrage et du contrôle des actionneurs, peut recevoir deux intentions concurrentes. Par exemple, chacun des asservissements va générer périodiquement un vecteur de commande (correspondant à leurs intentions) et le mécanisme d'arbitrage les reçoit afin d'en faire la pondération et la sommation, avant d'appliquer le vecteur de commande résultant aux actionneurs. Cela nécessite l'utilisation d'un protocole adapté et souvent complexe (e.g. protocole de vote dans AURA).

L'intégration d'une approche comportementale, si elle semble dans l'absolu souhaitable dans un souci de généricité du modèle, complique énormément la gestion des interactions (en plus d'amener d'autres facteurs de complexité déjà expliqués par ailleurs). Cette complexité est d'ailleurs un facteur qui fait que l'approche comportementale n'est que peu intégrée dans la plupart des propositions d'architectures (les modèles AURA et MAAM, par exemple, sont des exceptions notables). L'intégration des deux modèles d'organisation actuellement en concurrence (comportemental et délibératif) constitue en soit une problématique vaste allant au delà du travail réalisé durant cette thèse. Ainsi, si le modèle proposé dans cette synthèse laisse la possibilité à ces deux approches de coexister, nous ne développons pas cet aspect dans le reste du mémoire.

Besoins émergents

Ce modèle d'organisation d'une architecture de contrôle est suffisamment neutre pour pouvoir être étendu à un très grand ensemble de robots de services (robotique d'intervention terrestre ou sous-marine, domotique, etc.). Dans notre domaine d'application particulier qu'est la robotique mobile terrestre, les principes d'organisation présentés devraient être spécialisés afin de fournir un guide de conduite plus précis et intuitif. Nous pensons qu'un modèle plus détaillé permettrait de donner aux développeurs des concepts clairs, qui permettent de favoriser l'identification et la réutilisation des entités de contrôle et des entités de représentation des connaissances. Pour cela, il faudrait, en premier lieu, définir à partir de ce modèle générique, un modèle spécialisé qui mette en relief :

- le nombre de couches, et l'utilité de chaque couche,
- les différents types d'entités de contrôle présents dans chaque couche et leurs responsabilités respectives,
- les différents types d'interactions entre ces types d'entités de contrôle,

– les différents types d'entités de représentation des connaissances utilisées par les entités de contrôle.

3.8.5 Environnement et outils logiciels

Un autre constat de cette étude est qu'au delà d'un modèle d'organisation d'une architecture de contrôle, il est essentiel d'offrir les outils logiciels pour développer, analyser, simuler et exécuter des contrôleurs. Or bien souvent dans les approches présentées, les outils logiciels utilisés sont intimement couplés au modèle d'organisation d'architecture lui-même. L'avantage de cette approche est qu'elle fournit des outils très spécialisés (parfois même, comme dans le modèle du LAAS, en fonction des différentes couches). La faiblesse de cette approche est qu'elle manque de "souplesse" : dès lors que les développeurs veulent utiliser un nouveau modèle d'organisation, les outils logiciels utilisés jusqu'alors deviennent, au moins en partie, obsolètes. Ceci est un frein évident à la réutilisation et à la possibilité d'intégrer des entités logicielles initialement conçues pour des modèles d'organisation différents. Cette approche peut également rendre difficile la distinction des choix technologiques (langages et outils de programmation en particulier), des choix de conception (principes d'organisation en particulier), à partir desquels a été créée une architecture de contrôle.

Pour définir une méthodologie visant à rationaliser le développement de contrôleurs logiciels, il nous semble important de déterminer clairement les préoccupations associées à la plate-forme utilisée (au sens large : outils logiciels, langages de programmation, intergiciel et OS, etc.) et les préoccupations que doit intégrer le modèle d'organisation utilisé. Plus on associe la gestion de préoccupations à la plate-forme et plus ces préoccupations pourront être automatisées (ou au moins plus facilement prises en main) via l'utilisation d'outils spécifiques. L'approche du LAAS en est un bon exemple, car toutes les couches d'une architecture de contrôle sont développées avec des outils spécifiques, à tel point qu'il est possible de penser qu'organisation et outils sont indissociables. D'un autre côté, ceci entraine inévitablement la complexification de la plate-forme et peut limiter son utilisation au seul modèle d'organisation supporté.

Résumé :

Les architectures de contrôle mixtes (ou « hybrides ») :

- sont organisées suivant une approche hiérarchique :
  - une couche représente un « niveau de décision » et différents critères permettent de hiérarchiser les couches (priorité de réaction, contraintes temporelles de réaction, etc.).
  - une couche est constituée d'entités de contrôle.
  - une entité de contrôle obéit à un schéma « Perception – Décision - Action ».
  - une entité de contrôle manipule et échange des connaissances sur le monde du robot.
  - le transfert d'information peut se faire en « sautant » des couches.

- sont organisées suivant une approche systémique :
  - un système abstrait une partie du contrôleur et de la partie opérative d'un robot.
  - les systèmes sont décrits à partir d'entités de contrôle organisées suivant une approche hiérarchique et ils collaborent entre eux au niveau de différentes couches.

- sont basées à la fois sur des techniques délibératives (IA) et comportementales (arbitrage).

- nécessitent des outils :
  - pour programmer/exécuter les entités logicielles.
  - pour vérifier/valider (analyse formelle, simulation) le comportement du robot.
  - indépendants des modèles d'organisation conceptuels.

Chapitre 4

Composants logiciels

4.1 Introduction

Le paradigme de la conception et du développement d'applications logicielles à base de composants devient de plus en plus prégnant pour l'industrie du logiciel. Ceci peut s'expliquer par le fait que cette industrie tente continuellement de réduire le coût et le temps de développement des logiciels, en réutilisant au maximum ce qui a déjà été réalisé. Afin de faire évoluer significativement les possibilités d'échanges et de réutilisation des briques logicielles, la notion de composant sur étagère (COTS - Component Off the Shelf) a été proposée. Deux notions centrales et complémentaires émergent : les *composants logiciels* et les *architectures logicielles*.

Les composants logiciels sont les briques logicielles de base permettant la construction d'applications logicielles. Une définition classique [114] permet de qualifier plus précisémment ce qu'est un composant : *A software component is a unit of composition with contractually specified interfaces and explicit context dependencies only. A software component can be deployed independently and is subject to composition by third parties.* Cette définition, relativement générale, permet de qualifier l'objectif principal lié à l'utilisation des composants logiciels : il s'agit de réutiliser le code logiciel sous forme de briques de base, qui pourront ensuite être composées afin de construire des applications, par opposition à une reproduction systématique des briques logicielles à chaque nouvelle application.

Une architecture logicielle peut être définie de la manière suivante [99] : *The architecture of a software system defines that system in term of components. In addition to specifying the structure and topology of the system, the architecture shows the intended correspondance between the system requirements and elements of the constructed system. It can additionally address system-level properties such as capacity, throughput, consistency, and component composability. Architectural models clarify structural difference among components and interactions. Architectural definition can be composed to define larger systems [...]* Une architecture logicielle est une vision d'ensemble d'une application, qui peut être assimilée à un plan de construction, qui exprime la façon dont les composants interagissent.

L'idée de construire des application à base de composants s'est développée dans différentes communautés en parallèle (génie logiciel à objets, spécification de systèmes, informatique distribuée). Deux approches en ont émergé : les modèles de composants et les langages de desciption d'architecture (ADLs, Architecture Description Languages). Ces deux approches tentent de mettre en œuvre le paradigme des composants logiciels suivant deux points de vues différents mais complémentaires. Le point commun de ces deux approches, est d'avoir pour objectif de produire rapidement des applications de qualité, maintenables et évolutives, dans lesquelles différents composants peuvent être interchangés afin d'ajouter des fonctionnalités ou afin de corriger certaines erreurs.

4.1.1 Principes relatifs aux modèles de composants

Il existe actuellement de nombreux modèles de composants, tant issus des milieux académiques qu'industriels. Un modèle de composants décrit les concepts à partir desquels sont définis des composants logiciels, ainsi que différents mécanismes basé sur ces concepts :

- Un mécanisme de *composition* permet de mettre en relation au moins deux composants, afin qu'ils réalisent une ou plusieurs fonctionnalités d'une application.
- Un mécanisme d'*exécution* interprète le code des composants, afin que ceux-ci réalisent leurs fonctionnalités.
- Un mécanisme de *déploiement* permet de rendre des composants prêts à être exécutés.
- Un mécanisme de *communication* permet à au moins deux composants d'échanger des messages. Ces échanges sont définis en fonction de la composition des composants et de leur déploiement. Ils sont réalisés pendant leur exécution.

Chaque modèle de composants est sous-tendu (de manière explicite ou implicite) par une plate-forme qui implante les mécanismes qu'il définit. Dans le monde des composants distribués, cette plate-forme est appelée *intergiciel* (plus connue sous le terme anglais de *middleware*). La définition des concepts et mécanismes varie d'un modèle de composants à un autre, car chaque modèle de composant est dédié à un ensemble de préoccupations prédéfinies et différentes. Ceci a évidemment pour conséquence que des composants issus de modèles différents ne peuvent interopérer (i.e. fonctionner ensembles).

Les notions essentielles décrites dans les modèles de composants sont résumées dans la thèse de R. Marvie [83] :

- *Le type d'un composant*, c'est-à-dire sa définition abstraite, indépendante de ses différentes implantations potentielles. Un type est caractérisé par trois éléments : ses interfaces, les modes de coopération avec les autres types de composants et ses propriétés configurables. Les interfaces sont de deux sortes : *les interfaces fournies* qui permettent au composant d'offrir un ensemble de services et les *interfaces requises* qui représentent les services qu'utilise un composant. La notion d'interface requise permet de représenter explicitement les relations entre composants et facilite la substitution d'un composant par un autre. Chaque interface définit un mode de communication avec les autres types de composants comme par exemple le mode synchrone ou asynchrone. Les propriétés configurables décrivent les caractéristiques internes d'un composant qui peuvent être paramétrées.
- *L'implantation d'un composant*, qui regroupe son implantation *fonctionnelle* et son implantation *non-fonctionnelle*. L'implantation fonctionnelle d'un composant représente sa mise en œuvre d'un point de vue métier, indépendamment des conditions d'exécution. Cette implantation fonctionnelle est rattachée à l'ensemble des interfaces fournies et requises, ainsi qu'aux propriétés configurables, définies au niveau du type. L'implantation non-fonctionnelle représente l'adaptation de ce code métier aux conditions d'exécution.
- *Le paquetage d'un composant*. Un paquetage d'un composant est une entité déployable (souvent une archive) contenant son type, au moins une de ses implantations, ainsi que les contraintes techniques associées à chaque implantation (langage de programmation utilisé, OS cible, etc.).
- *L'instance d'un composant*. Une instance d'un composant est une entité s'exécutant dans une application. Chaque instance est caractérisée par un identifiant unique, un type de composant et une implantation particulière de ce type. Une instance d'un composant possède une valeur pour chaque propriété configurable définie au niveau de son type.

Une des particularités des composants logiciels, par rapport aux objets, vient du mécanisme utilisé pour les composer. Notons que nous utilisons le terme générique de *composition* pour désigner un mécanisme permettant de mettre en relation des briques logicielles (objets ou composants) afin qu'elles interopèrent (et non pas le sens donné à ce terme dans UML). Dans la modélisation et la programmation objet, les objets sont composés en fonction des comportements décrits par leurs classes. Le code d'un objet (attributs et méthodes) contient des références directes vers d'autres

objets. A la différence des objets, les composants logiciels ne possèdent pas de références directes sur d'autres composants, mais uniquement sur des interfaces. Un type de composant est défini sans connaître les types avec lesquels il est suceptible d'être composé, mais juste en fonction des interfaces qu'il fournit et requiert. La composition de deux composants n'est pas définie dans leur code, mais effectuée par un *tiers* (i.e. le "third-party", par exemple un développeur) qui établit une relation entre leurs interfaces fournies et requises. C'est ce mécanisme de composition, nommé *assemblage*, qui permet aux composants d'avoir un pouvoir de réutilisation a priori supérieur à celui des objets.

4.1.2 Principes relatifs aux langages de description d'architectures

Les langages de description d'architecture (ADLs, Architecture Description Languages) sont un support pour la modélisation d'architectures logicielles, décrites par des assemblages de composants logiciels. Il existe de nombreux ADLs, comme C2 [85], Rapide [79], MetaH [24], Wright [6], Weaves [62], dont les objectifs sont variés : description formelle de l'architecture logicielle, génération automatique du code "squelette" d'une architecture logicielle, intégration et déploiement (semi-)automatique des composants, etc. Les principales caractéristiques des langages de description d'architectures ont été définis par N. Medvidovic [86] :

Le but minimal d'un ADL est la modélisation d'une architecture logicielle grâce à laquelle les développeurs possèdent une documentation qui leur permet de mieux comprendre l'application développée. Cette modélisation est faite à partir d'entités que sont les composants, les connecteurs et les configurations :

- Un *composant* est une unité de stockage et/ou de traitement des données. Il possède plusieurs propriétés, nous en retiendrons quatre : son type, ses interfaces, sa sémantique, ses contraintes. Son type déclare les caractéristiques communes partagées par un ensemble de composants. Chacune de ses interfaces déclare des services qu'il fournit et/ou requiert. Sa sémantique décrit son comportement "attendu" (i.e. ce qu'il doit faire) à l'exécution. Chacune de ses contraintes exprime une règle qu'il doit respecter.
- Un *connecteur* est une entité modélisant un protocole d'interaction et qui sert à assembler des composants. Un connecteur peut modéliser tout aussi bien un appel de procédure à distance (RPC, Remote Procedure Call), qu'un protocole de communication sécurisé et transactionnel. Dans de nombreux ADLs, un connecteur est une entité de première classe (il est décrit, manipulé et réutilisé explicitement au même titre qu'un composant). Un connecteur possède des propriétés similaires à celles d'un composant, dont nous retiendrons les quatre plus importantes : son type, ses interfaces, sa sémantique, ses contraintes. Son type déclare les caractéristiques communes d'un ensemble de connecteurs. Chacune de ses interfaces décrit un rôle joué par un composant, dans le cadre de l'interaction qu'il décrit. Un rôle est une spécification des services qu'un composant doit fournir et/ou requérir pour participer à l'interaction. Sa sémantique décrit le protocole d'interaction entre les rôles. En général, un ADL offre la même notation pour exprimer la sémantique des composants et des connecteurs, afin d'avoir une sémantique homogène, autorisant l'analyse des assemblages de composants. Chacune de ses contraintes permet d'exprimer une règle d'assemblage qui s'applique aux composants jouant les rôles qu'il définit.
- *Configuration.* Une configuration décrit une architecture logicielle, elle agrège un assemblage de composants connectés par des connecteurs. Elle permet de manipuler un assemblage de façon unitaire, comme si c'était un composant. Cela autorise la description d'une architecture logicielle à plusieurs niveaux d'agrégation (une configuration peut contenir des configurations), ce qui facilite la lisibilité, et donc la compréhension de l'architecture. Une configuration possède comme propriétés un type et des interfaces, au même titre que les composants. Elle peut, suivant les ADLs, posséder en plus des propriétés permettant de décrire le déploiement ou les performances attendues de l'architecture qu'elle contient.

Medvidovic et Taylor [86] classifient les ADLs en trois grandes catégories.

- Les ADLs de configuration, comme C2 [85], permettent de décrire des architectures contenant les informations nécessaires pour générer le code squelette de leurs composants et connecteurs. Certains permettent de manipuler des informations utiles au déploiement et à l'administration des applications instances de ces architectures.
- Les ADLs d'intégration, comme ACME [58], permettent d'intégrer et de partager des éléments d'architecture définis via des ADLs hétérogènes.
- Les ADLs formels reposent sur des formalismes de modélisation qui permettent de décrire les propriétés des composants, des connecteurs et des configurations (e.g. leur comportement, leurs contraintes d'utilisation, leurs propriétés non-fonctionnelles, etc.). L'utilisation de notations formelles permet aux développeurs d'utiliser des outils d'analyse afin de déduire d'éventuelles erreurs de conception (ou ambiguïtés) à partir d'une description. La validation et/ou la vérification des spécifications architecturales est centrale dans ces langages.

Nous nous intéressons plus particulièrement à cette dernière catégorie d'ADLs, dont les caractéristiques nous semblent fondamentales au développement de contrôleurs. Le développement de ce genre d'applications nécessite, en effet, des méthodes de vérification/validation qui permettent d'éviter, dans une certaine mesure, des erreurs de conception. La suite de ce chapitre est composée de deux sections. La première présente différents modèles de composants et langages de descriptions d'architectures. La seconde propose un rapprochement entre les propositions des ADLs et des composants logiciels.

4.2 Propositions sur les composants logiciels

4.2.1 Darwin

Darwin est un modèle de composant précurseur [80] proposé en 1994 par le Distributed Software Engineering Group de l'Imperial College de Londres. Il est dédié à la construction d'applications réparties [81].

Dans le modèle Darwin, un composant est une unité de traitement de l'information qui possède une interface qui déclare les ports qu'il fournit et requiert. Ces ports correspondent respectivement aux flots d'entrée et de sortie des composants. Les ports n'ont pas de connotation fonctionnelle, ils désignent seulement les points de communication du composant avec le monde extérieur, via lequel les opérations des composants sont appelées. Chaque port est associé à un mode de communication (e.g. synchrone, asynchrone). Il existe deux sortes de composants : les primitifs et les composites. Un composant primitif encapsule du code implantant la fonctionnalité métier qui lui a été attribué. Un composite est une unité de configuration qui contient la description d'interconnexions d'instances de composants primitifs et de composites. Une application est donc un composite.

La figure 4.1, tirée de [81], donne un exemple de composants Darwin. Le composant primitif, nommé `Filter`, est un filtre dont l'interface est composée d'un port d'entrée `left` qui récupère un entier. Si cet entier correspond au critère du filtre, ce composant le transmet alors par le port de sortie `output`, sinon il est transmis par le port de sortie `right`. Le composite, nommé *Pipeline*, est une composition d'un ensemble de `Filter`. Il chaine les filtres les uns avec les autres. Un composite permet de décrire une architecture dynamique : le nombre et la nature des composants qu'il agrège, ainsi que leurs interactions, peuvent être paramétrés. Par exemple, le composite `Pipeline` décrit une architecture dont le nombre de composants `Filter` peut être paramétré (en fonction de l'entier `n`).

Darwin est fourni avec un support d'exécution des composants et de communication inter-composants. Les composants primitifs sont implantés par des classes C++ héritant d'une classe

```
//déclaration d'un composant primitif
Component Filter(){
      //partie interface
      provide left; //port fourni
      require right; //port requis
      require output;
      //partie implémentation vide, l'implantation est séparée dans une classe C++
}

//déclaration d'un composite
Component Pipeline(int n){
      //partie interface
      provide input;
      require output;
      //partie implémentation composée de déclaration d'instances de composants et de
schémas          //d'interconnexion (cas d'un composite)
      array F[n] Filter; //tableau contenant l'ensemble d'instances de filtres
      forall k : 0 ... n-1{
            inst F[k] //déclaration d'une instance de filtre

            bind F[k].output - output;
            //lien en la sortie output du filtre courant et la sortie output du Pipeline

            when k < n-1
                  bind F[k].right - F[k+1].left;
            //lien entre les ports droit et gauche des filtres afin de les chaîner
      }
      //lien entre le service left du premier filtre et le service input du Pipeline
      bind input - F[0].left;
      //lien entre le service right du dernier filtre et le service output du Pipeline
      bind F[n-1].right - output;
}
```

FIG. 4.1 – Exemple d'une application décrite avec Darwin, tiré de [81]

"Process". L'instanciation d'un composant primitif correspond à la création d'un processus système. Le support d'exécution de Darwin, permet également d'interpréter le code d'implantation des composites, ce qui consiste, entre autre, à configurer les communications des composants primitifs. L'instanciation d'un composite correspond, in fine, à la création d'un ensemble de processus systèmes communicants.

Darwin a contribué à la définition de concepts importants dans les approches par composants. Tout d'abord, le concept de *port*, permet de décrire un point d'interconnexion d'un composant, via lequel il communique suivant un mode de communication donné. Les concepts de port *requis* et *fournis* sont également importants, car ils permettent de déclarer explicitement ce dont un composant a besoin pour réaliser ses fonctionnalités et ce qu'il permet de réaliser. Grâce à ces concepts, il est possible de mettre explicitement en relief les liens entre composants d'une architecture. Une limitation de Darwin est de ne pas fournir de moyen de définir de nouveaux modes de communication, car ceux-ci sont prédéfinis au sein du support d'exécution. Enfin, le concept de *composite* permet de structurer les architectures en différents niveaux d'agrégation, ce qui est d'importance pour des architectures logicielles de grande taille.

4.2.2 Java Beans

Le modèle des Java Beans [67] a été proposé par Sun Microsystems en 1996. Son premier et principal domaine d'application est la construction d'interfaces graphiques, mais son utilisation peut être étendue à d'autres domaines.

Suivant la définition de Sun Microsystems, un Java Bean est un composant logiciel réutilisable qui peut être manipulé visuellement à travers un outil logiciel. Un Java Bean possède une interface

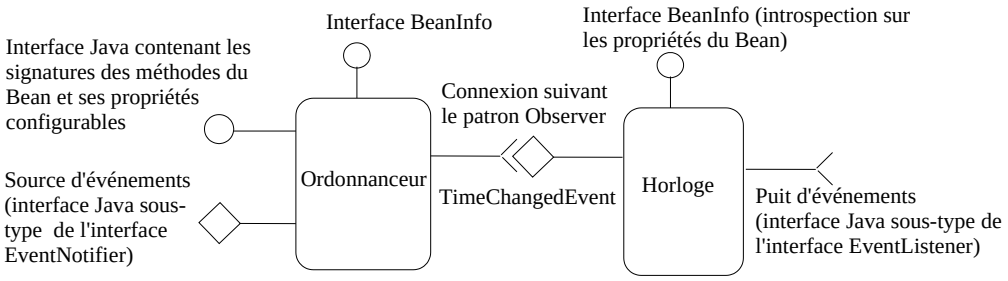


FIG. 4.2 – Modèle des Java Beans

qui décrit ses méthodes et ses attributs. La composition des instances repose sur la notification et l'écoute d'événements (fig. 4.2). Chaque Bean écoute (chez d'autres Bean) un ensemble d'événements auxquels il peut réagir et notifie ses écoutants d'un ensemble d'événements qu'il produit. Chaque type d'événement auquel il sait réagir est représenté par un puit d'événements. Chaque type d'événement qu'il sait produire est représenté par une source d'événements. Au niveau de son interface un Bean est identique à un objet Java ce qui implique qu'il peut être utilisé comme tel. Un Bean possède également un mécanisme standard d'introspection qui permet, à travers son interface `BeanInfo`, d'obtenir des informations sur lui-même (ses propriétés configurables, les différents types d'événements écoutés ou notifiés, ses différents services).

La programmation d'un Java Bean se fait de manière analogue à celle d'une classe Java, mais certains mécanismes spécifiques sont définis en fonction de patrons de conception [57] et de règles de nommage. Chaque Beans peut posséder des propriétés, qui sont définies par la présence d'une ou deux opérations permettant leur lecture (opération `get`) et éventuellement leur modification (opération `set`). La définition de ces propriétés suit le patron de conception *Accesseur*. Le mécanisme de notification d'événements suit le patron de conception *Observer*. Une source d'événements correspond à un envoi, par le Bean, d'événements d'un type donné. Chaque source d'événements est traduite par une opération d'abonnement et une opération de désabonnement, associées à l'écoute d'un type d'événement considéré. Un puit d'événements est traduit par une interface de consommateur d'événements (`Eventlistener` ou une de ses sous-interfaces), qui définit une opération de réaction à un événement particulier et qui prend en paramètre un objet implantant l'interface `Event` (ou une de ses sous-interfaces). Les classes implantant l'interface `Event` contiennent toutes les informations et opérations décrivant un type d'événement. Ce mécanisme de notification d'événements possède également une spécialisation notable : la possibilité de rendre impossible la modification des valeurs des propriétés d'un Bean. Un Bean peut permettre à d'autres Beans de s'enregistrer à l'écoute d'événements traduisant la modification d'une propriété. Les Beans consommateurs de ces événements peuvent donner leur veto à une modification de ses propriétés (la modification ne se fait donc pas). Un Java Bean étant décrit par une classe Java, il peut implanter plusieurs interfaces (typiquement, une par propriété du Bean). Ce mécanisme, qui suit le patron de conception *Adapter* permet de séparer les différentes interfaces d'un Bean de son implantation.

L'utilisation de ces patrons et de conventions de nommage, couplé au mécanisme d'introspection via l'interface `BeanInfo`, permet à des outils d'édition spécialisés de prendre connaissance de la structure d'un Bean à l'exécution. De tels outils peuvent ainsi fournir une représentation visuelle d'un Bean (propriétés et événements produits ou écoutés) et diriger facilement l'interconnexion des Beans

(lien entre puits et sources d'événements, accès aux propriétés). L'environnement d'exécution des Java Beans est une machine virtuelle Java standard, couplée à un conteneur graphique, tel qu'un navigateur Web, dans le cas d'un Bean graphique. Cela donne des facililtés pour la diffusion et le déploiement des Java Beans (sous forme d'archive Java). L'utilisation exclusive de Java apporte une forte portabilité aux Java Beans, mais limite évidemment leur utilisation au langage de programmation Java.

Le modèle des Java Beans a contribué à l'évolution de la notion de composant sur étagère. Le modèle de communication par événements a révolutionné la façon d'assembler des composants logiciels. Cette forme de composition permet à des Java Beans n'étant pas initialement conçus pour fonctionner ensemble, d'être composés a posteriori : il suffit que l'un soit producteur et l'autre consommateur d'un même type d'événement. Pour faciliter un assemblage a posteriori, il faut évidemment posséder des types d'événements standards à partir desquels les Java Beans sont définis.

4.2.3 Enterprise Java Beans

Le modèle de composant Enterprise Java Beans (EJB) [45] a été développé en 1997 par Sun Microsystems. Il est dédié au développement, à la gestion et à l'évolution de composants serveurs au sein d'applications distribuées 3-tiers (clients/serveurs/bases de données). Dans ces applications les aspects non-fonctionnels, tels que les aspects transactionnel, persistance et sécurité, sont très importants.

Un composant EJB (fig. 4.3) est une instance d'une classe java de type `EJBClass`. Une `EJBClass` implante un ensemble d'interfaces spécifiques :

- l'interface *Remote* déclarant les services dits "métiers" que peut rendre l'EJB à un client (chaque service correspondant à une opération). La déclaration de l'interface *Remote* d'un EJB se fait au travers d'une interface Java qui étend l'interface `EJBObject`.
- l'interface *Home* déclare un ensemble de services standards permettant de gérer le cycle de vie du composant (création et destruction d'instances, recherche d'un EJB persistant). Ces services sont utilisés par les administrateurs de l'application distribuée. La déclaration de l'interface *Home* d'un EJB se fait au travers d'une interface Java qui étend l'interface `EJBHome`.
- L'interface de meta-données permet, à l'exécution, de connaître les meta-données associées à l'EJB. Ces meta-données représentent son interface *Remote*, ses attributs, etc. Ceci permet, par exemple, de construire dynamiquement des requêtes sur l'EJB en fonction des services déclarés dans son interface *Remote*. L'interface de meta-données d'un EJB est accessible à un administrateur via l'interface *Home*.

La classe d'un EJB spécialise un des trois types d'EJB proposés : session, entité et message. Lorsqu'une classe est de type session (`EJBSession`), cela veut dire qu'une session sera ouverte à la connexion d'un client, ce qui correspond à créer un EJB pour chaque connexion client et à le détruire à la demande de ce client uniquement. Lorsqu'une classe est de type entité (`EJBEntity`), ses instances sont vues comme étant persistantes. Un identifiant unique est associé à chaque EJB créé, ce qui permet des recherches d'instances par exemple. Le cycle de vie d'un EJB entité n'est pas soumis à celui d'un client. Lorsqu'une classe est de type message (`EJBMessage`) ses instances ne sont pas persistantes, ni soumises au cycle de vie d'un client, et peuvent interagir avec plusieurs clients (de façon asynchrone). L'implantation d'un EJB fait référence à un ensemble de services non-fonctionnels dont il a besoin pour s'exécuter (par exemple, besoin du service de persitance).

Chaque EJB possède un descripteur de déploiement qui contient des informations diverses sur le composant (e.g. fournisseur, version) et permet de décrire l'ensemble des services non-fonctionnels qu'il utilise. Pour être exécuté, chaque EJB doit être hébergé dans un *conteneur* : c'est une entité exécutable sur un serveur d'application, qui définit la façon d'utiliser des services non-fonctionnels pour un contexte applicatif donné. La prise en charge des services non-fonctionnels se fait donc au

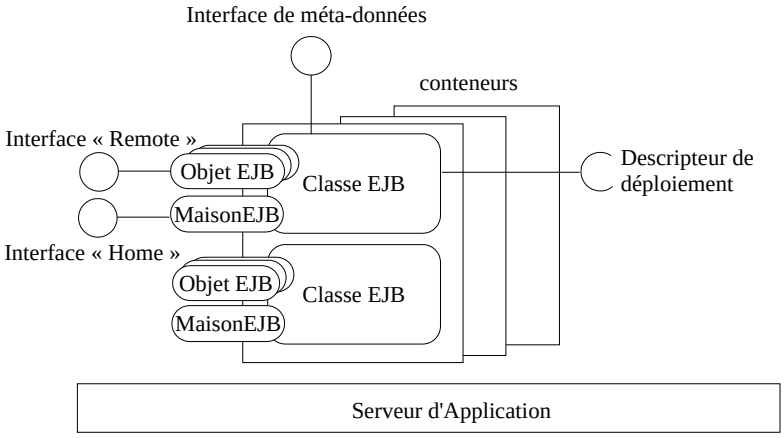


FIG. 4.3 – Modèle des composants EJB

moment de la création des *conteneurs*. Pour pouvoir réaliser sa tâche, un *conteneur* intercepte les messages entrants et sortants des EJB qu'il héberge. En fonction de ces messages il applique des appels aux services non-fonctionnels fournis par le serveur d'application. Un descripteur de déploiement permet de configurer les actions du conteneur en fonction de certains services non-fonctionnels (e.g.persistance, sécurité). Un serveur d'application est un intergiciel supportant l'exécution d'un ensemble de *conteneurs*. Plusieurs serveurs d'application peuvent être utilisés pour le déploiement d'une application distribuée sur plusieurs sites.

Le modèle EJB apporte des concepts innovants à la notion de composant, essentiellement autour de la gestion de leur déploiement. La gestion des propriétés non-fonctionnelles à travers la notion de *conteneur* fut une idée novatrice. Les besoins d'un EJB en termes de services non-fonctionnels sont exprimés via un descripteur qui est utilisé au moment du déploiement pour configurer (en partie) le *conteneur* qui l'héberge. Un *conteneur* permet de configurer l'utilisation des services non-fonctionnels par les EJB, en adaptant leurs besoins non-fonctionnels, aux ressources réellement présentes sur le serveur d'application. Ce découpage entre code métier et utilisation des services non-fonctionnels permet d'augmenter sensiblement la réutilisabilité des composants EJB.

La limitation forte du modèle EJB découle du fait que les connexions des composants EJB ne sont pas visibles au niveau de l'architecture logicielle. Elles sont "enfouies" dans le code de chaque EJB, comme dans le modèle Objet, car le concept de port (ou d'interface) requis est absent du modèle. Puisque la connectique d'un composant EJB est gérée dans son implantation, la modification de l'architecture logicielle est beaucoup plus complexe que dans les approches intégrant la notion de port requis.

4.2.4 CORBA Component Model

Le modèle de composant CORBA (CCM) est le fruit d'une longue maturation des approches par composants. Il est issu de la dernière norme CORBA 3.0 [90]. Les normes CORBA précédentes ont permis la spécification d'un intergiciel pour objets distribués qui supporte l'interopérabilité entre des objets distants et hétérogènes (au sens des langages de programmation et des systèmes d'exploitation) et d'une méthodologie de développement associée [60]. C'est à partir de la création de cette norme que le terme de composant logiciel a commencé à prendre de son importance. La première norme a

notamment mis en avant l'importance de la notion d'interface et la prise en compte des différentes implantations d'un composant. La complexité des applications réparties actuelles a conduit la norme CORBA a évoluer vers un modèle de composant complet, s'inspirant de ce qui a été proposé par d'autres consortiums ou industriels, comme les EJBs, COM (Component Object Model)/DCOM (Distributed Component Object Model) de Microsoft [63] et ODP [72] de l'ISO. A l'heure actuelle, le CCM semble être le modèle le plus complet du marché, car il regroupe la plupart des points forts des modèles concurrents et tire parti de l'intergiciel CORBA.

Le modèle de composants CCM s'appuie sur deux notions fondamentales héritées des objets distribués CORBA : la notion d'intergiciel et le langage IDL (Interface Definition Language). Le langage IDL permet de décrire des interfaces (attributs et opérations) qui sont indépendantes de tout langage de programmation et qui supportent les mêmes mécanismes (héritage en particulier) que les interfaces des objets. Les interfaces IDL servent à décrire des contrats liant les composants clients aux composants serveurs. L'intergiciel CORBA, nommé ORB (Object Request Broquer), est une plate-forme fournissant un support technique à l'exécution des composants et la prise en charge transparente de leurs communications. L'ORB permet de faire abstraction des moyens de communication réseau ou système utilisés au moment du développement des composants. Une interface IDL peut-être projetée (cf. fig. 4.4) vers différents langages de programmation, suivant des "mapping" définis par l'OMG. Cette projection amène la création d'objets CORBA dans le langage de programmation choisi. Un objet CORBA généré du côté serveur est appelé *squelette* et *souche* lorsqu'il est généré du côté client. Le squelette est l'objet serveur qui possède une référence vers l'objet métier qui implante les méthodes et attributs déclarés au niveau de l'interface IDL. La souche est l'objet client qui a une référence vers un objet squelette, et qui permet à l'application cliente d'interroger l'objet serveur distant de la même manière qu'avec des objets classiques. Souches et squelettes s'exécutent au-dessus d'un ORB qui leurs fournit les moyens de communiquer grâce au protocole standard GIOP (General Inter-ORB protocol). Ce protocole générique peut être spécialisé pour différentes couches de transport, comme par exemple le protocole IIOP (Internet inter-ORB protocol) qui est la version TCP/IP de GIOP. L'intergiciel CORBA étant spécifié à partir d'interfaces IDL, il est lui aussi sujet à diverses implantations pour des langages et/ou des systèmes d'exploitation différents. Le mécanisme proposé par la norme CORBA 1.0 et les suivantes amène des avantages tels que l'interopérabilité technique entre différents langages de programmation et/ou OS ainsi que la prise en charge transparente des communications entre objets.

La norme CORBA 3.0 fournit une extension du langage IDL 2.0 (nommée IDL 3.0), qui permet la description d'un composant CCM (cf. fig. 4.5). Dans le CCM, la partie structurelle d'un composant est décrite intégralement à partir d'interfaces écrites en IDL 3.0. IDL 3.0 est traductible dans le langage IDL 2.0, ce qui permet aux composants CCM de profiter pleinement de tous les avantages des objets distribués CORBA cités précédemment. Le modèle CCM repose sur deux points essentiels : le premier est qu'un composant CCM est une boite noire dont on ne sait rien de l'implantation ; le second que la structure du composant doit être entièrement décrite en IDL 3.0, alors que son implantation fonctionnelle doit être programmée. Le but sous-jacent est de séparer clairement le type du composant de son implantation.

Le type d'un composant définit sa structure en terme d'interfaces fournies (i.e. que le composant implante) et d'interfaces requises (i.e. dont le composant requiert l'utilisation). A chaque interface, fournie ou requise, correspond un port qui définit le mode de communication associé à cette interface. Le langage IDL 3.0 définit un ensemble de types de ports différents : *provides*, *consumes*, *uses*, *emit* et *publishes*. A chaque type de port correspond donc un mécanisme de connexion et de communication spécifique.

- Une *facette* est une interface fournie associée à un port de type *provides*, qui propose un mode de communication synchrone. Une facette regroupe un ensemble de signatures d'opérations. Une même opération peut être appelée via différentes facettes. Les facettes sont utiles à la

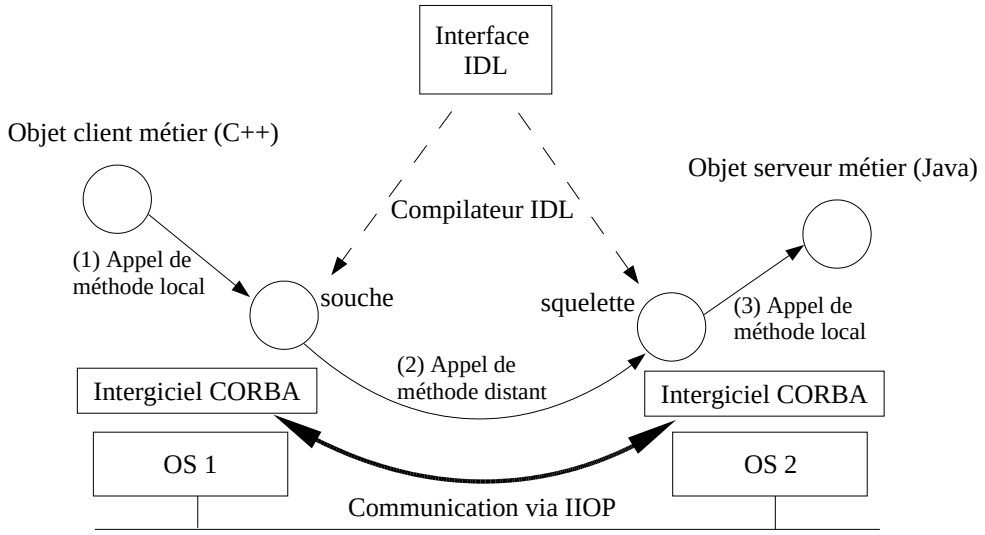


FIG. 4.4 – Intergiciel du CORBA

définition de points de vue sur le type de composant, un point de vue correspondant à un ensemble d'opérations utilisables par des clients spécifiques. Si par exemple, le composant est un système d'exploitation, il fournit une facette pour ses clients utilisateurs et une autre pour ses clients administrateurs.

- Un *puits d'événements* est une interface fournie associée à un port de type *consumes*, qui propose un mode de communication asynchrone par échange d'événements, du même type que celui proposé dans les Java Beans. Le type de composant déclare à travers un puit d'événements, pouvoir réagir à la notification d'un type d'événement donné.
- La *référence de base* est l'interface fournie (unique) qui permet d'identifier le type de composant et qui donne les moyens aux clients du composant de naviguer entre ses facettes et puits d'événements. Au niveau de l'interface de base sont décrits les attributs configurables associés au type de composant, qui peuvent être paramétrés par les clients.
- Un *réceptacle* est une interface requise associée à un port de type *uses*, sous-tendu par un mode de communication synchrone entre le composant et une facette d'un autre composant.
- Une *source d'événements* est une interface requise associée à un port de type *emit* ou *publishes*, qui traduit un envoi de messages asynchrones respectivement *1-vers-1* (communication privée entre instances) ou *1-vers-n* (diffusion d'un événement à un ensemble d'écouteurs).

Pour chaque type de composant est défini un (ou plusieurs) type(s) de maisons de composants. Une maison de composants est un gestionnaire qui contrôle le cycle de vie d'un ensemble d'instances (création / destruction / recherche) pour un type de composant donné. Implanter un composant CCM consiste à programmer les opérations déclarées dans ses facettes et puits d'événements. Afin de permettre une bonne intégration des parties fonctionnelles avec les parties non-fonctionnelles, le CCM propose le *Component Implementation Framework* (CIF). Le CIF spécifie la façon dont ces parties doivent coopérer, ainsi que la façon dont doit être générée la partie non-fonctionnelle. Pour cela, le CIF s'appuie sur le langage CIDL (*Component Implementation Definition Language*) qui permet de décrire la partie non-fonctionnelle d'un composant. Les informations décrites en CIDL portent sur le type d'implémentation (session, service, processus ou entité, de la même manière

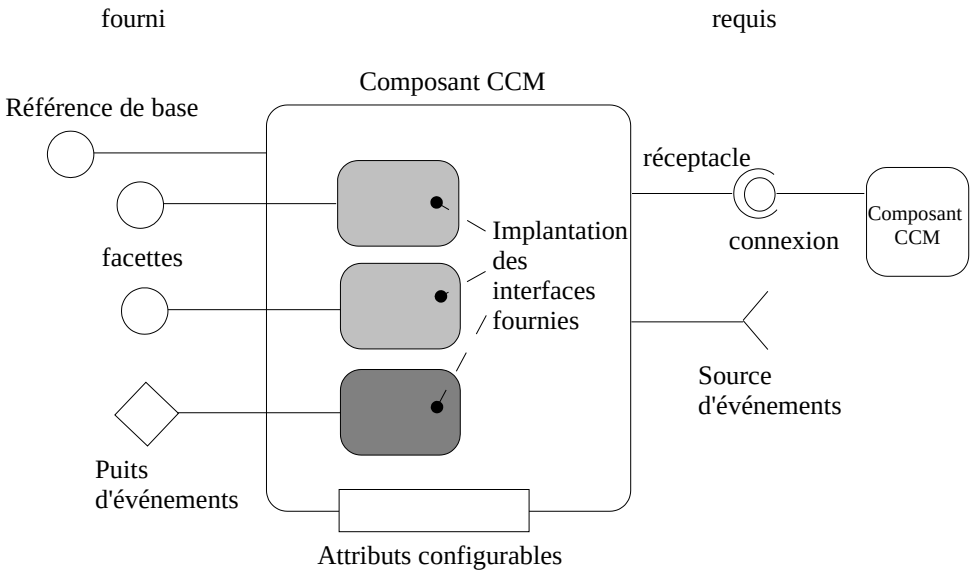


FIG. 4.5 – Le modèle de composant CCM

qu'EJB) ou la gestion de sa persistance. Cette description donne lieu à une génération automatique du code non-fonctionnel et à une intégration de ce code avec les parties fonctionnelles du composant. Comme dans les EJB, un composant CCM s'exécute dans un conteneur qui lui offre un environnement d'exécution (espace mémoire, flot d'exécution) ainsi que l'accès aux services non-fonctionnels qui lui sont nécessaires. Ces conteneurs s'exécutent au dessus de l'intergiciel de CORBA. Les échanges entre instances de composants et conteneurs sont définis par des API standardisées. Un conteneur permet, comme dans les EJB, d'adapter les appels des composants qu'il héberge aux services non-fonctionnels disponibles sur un ORB.

Le modèle CCM permet également de décrire le déploiement d'un composant, d'un package de composant ou d'un assemblage de composants, via l'utilisation d'un ensemble de descripteurs de déploiement. L'utilité de ces descripteurs est de permettre la diffusion et le déploiement automatique de tout ou partie d'applications. Il existe plusieurs sortes de descripteurs :

- La description d'un type de composant est réalisée à l'aide de descripteurs de composants, appelés CCD (*CORBA Component Descriptor*). Un CCD décrit les caractéristiques principales du type de composant (interfaces, ports, etc.) ainsi que les contraintes techniques liées à son implantation (langage de programmation, système d'exploitation, etc.) et ses propriétés non-fonctionnelles (services non-fonctionnels utilisés).
- Un descripteur de propriété, (CPF, pour *Component Property File*) définit un état initial d'un composant, qui précise les valeurs des attributs à fixer à la création d'une instance de composant.
- La description d'un assemblage de composants est réalisée à l'aide d'un descripteur d'assemblage, appelé CAD (*Component Assembly Descriptor*). Un CAD contient les éléments de description de chaque composant (CCD et CPF). Il contient la description du partitionnement de l'architecture en terme de maison de composants et les instances associées à chaque maison. Il contient la description de la manière dont les instances de composants sont connectées en fonction de leurs ports.

Le modèle de composant CCM est, à notre connaissance, le modèle le plus complet proposé à ce jour, d'un point de vue technologique. Il a permis en particulier de capitaliser la plupart des avancées précédentes dans le domaine des composants logiciels :

- Le CCM offre une clarification du concept de port. Un port est un point de connexion d'un composant, défini à partir d'une interface et d'un *mode de communication*. Les interfaces permettent d'exprimer les services fournis ou requis par les ports. Le mode de communication associé à un port peut être vu comme le protocole d'interaction utilisé pour connecter ce port. La limitation est que les modes de communication sont figés aux deux modes classiques d'interaction (interaction par notification d'événements et interaction par envoi de requête) comme c'est le cas par exemple dans les Java Beans, les développeurs ne pouvant en définir de nouveaux.
- Le CCM offre la possibilité de gérer les aspects non-fonctionnels indépendamment des aspects fonctionnels. La notion de conteneur comme descripteur d'un contexte d'exécution pour un ensemble de composants, permet d'adapter les aspects non-fonctionnels d'un composant (appels aux services offerts par l'intergiciel) à son contexte applicatif, système et matériel. Une limitation est que les services "techniques" utilisés sont figés à ceux proposés par dans la spécification CORBA ou par le fournisseur d'un ORB. Ces services devraient pouvoir être adaptables, évolutifs et éventuellement résulter de la composition d'autres services "techniques".
- Le CCM permet de décrire l'architecture d'une application comme un assemblage de composants, grâce aux différents types de descripteurs de déploiement proposés. Cette description va servir de support au déploiement (semi-)automatique de tout ou partie d'une application.
- Le CCM permet la construction d'applications à base de composants hétérogènes. Ceci est possible car le CCM est une norme issue d'un organisme de standardisation et non un modèle dépendant d'une technologie, comme les EJB. Actuellement, de tels environnements sont rares, ceci pouvant s'expliquer par la complexité de mise en œuvre de la spécification CORBA. Des exemples notables de travaux sur l'implémentation du CCM sont le projet OpenCCM [84] ou les travaux réalisés autour d'une implantation temps-réel du CCM à l'aide du framework TAO [125].

4.2.5 Fractal

Fractal [42] [31] est un modèle de composant reposant sur une spécification proposée par le consortium Object Web en 2002, regroupant chercheurs et industriels. Le modèle Fractal repose sur une spécification qui peut recevoir différentes implantations. Cette spécification est faite en langage naturel et dans un langage de description d'interface (IDL) très simple, qui peut être traduit dans divers langages comme Java et OMG IDL. D'autre part cette spécification se veut souple, en ce sens que presque toutes les propriétés décrites par le modèle sont optionnelles. Ainsi le modèle Fractal ne peut être mis en œuvre que partiellement, lorsque les contraintes techniques, liées à sa mise en œuvre sur une plateforme particulière, l'imposent. Grâce à cette souplesse tant au niveau du modèle abstrait que de son implantation, Fractal est un modèle "fédérateur" pouvant, a priori, être utilisé pour n'importe quel domaine applicatif.

Un composant Fractal possède un *contenu* et un *contrôleur* (fig. 4.6). Le *contenu* est constitué d'un ensemble fini d'autres composants, alors appelés sous-composants, qui sont sous le contrôle de son *contrôleur*. Le modèle des composants Fractal est récursif et permet donc la construction d'architectures à différents niveaux d'agrégation. Le contenu d'un composant contient son code "métier", alors que son contrôleur contient le code relatif à la gestion de ses aspects non-fonctionnels (e.g. transactionnel, persistance, etc.) ainsi que celui relatif à la gestion du cycle de vie de son contenu. Les composants ne sont pas obligatoirement des boites noires et peuvent rendre leur contenu visible depuis un super-composant (par opposition à sous-composant).

Le contrôleur est la partie du composant prenant en charge son interaction avec le monde exté-

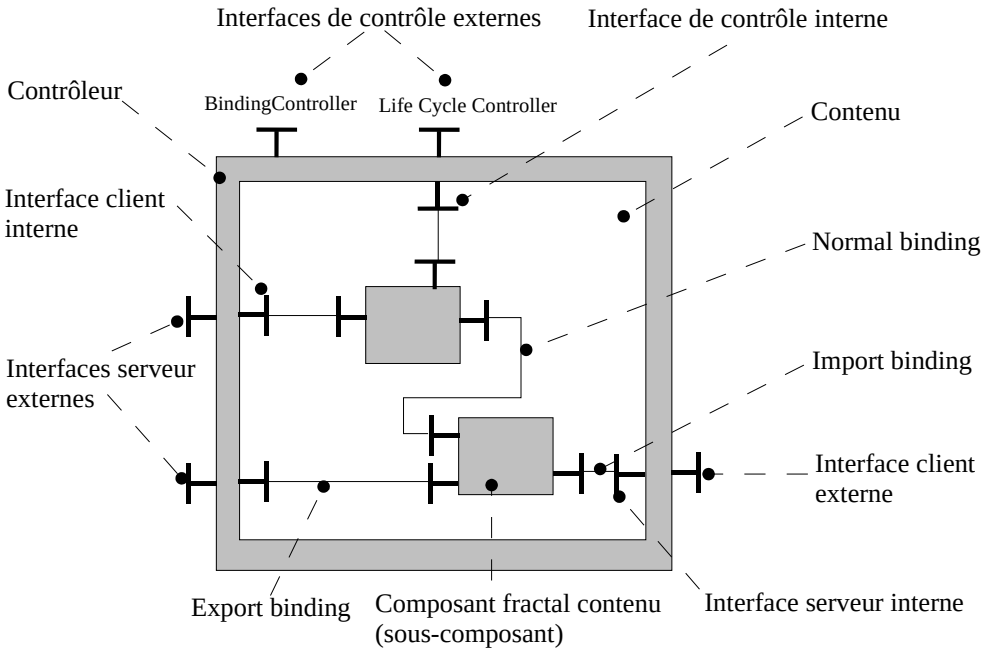


FIG. 4.6 – Le modèle des composants Fractal : exemple d'un composite

rieur. Il a une partie interne et une partie externe. Ses deux parties déclarent un ensemble d'interfaces, une interface y est définie comme étant un point d'accès au composant. La partie externe d'un contrôleur déclare un ensemble d'interfaces fonctionnelles dites *serveur* (i.e. fournies) et un ensemble d'interfaces fonctionnelles dites *client* (i.e. requises). Une interface *serveur* déclare un ensemble d'opération, alors qu'une interface *client* déclare un ensemble d'invocations d'opérations. Les interfaces fonctionnelles contiennent les opérations "métier" fournies ou requises par le composant. La partie externe d'un contrôleur déclare également des interfaces de contrôle internes, qui permettent de gérer les aspects non-fonctionnels et le cycle de vie du contenu. Il existe un ensemble standard d'interfaces de contrôle : *AttributeController*, *ContentController*, *BindingController*, *LifeCycleController*. L'interface *AttributeController* définit un ensemble d'opérations pour modifier à l'exécution la valeur des attributs d'un composant. L'interface *ContentController* définit un ensemble d'opérations pour modifier son contenu, c'est-à-dire consulter des composants par introspection, ajouter ou enlever des composants, partager des sous-composants. L'interface *BindingController* définit les opérations permettant d'assembler/désassembler (bind/unbind en anglais) les sous-composants du contenu. Enfin puisque ces opérations sont potentiellement périlleuses à l'exécution, une interface *LifeCycleController* définit un ensemble d'opérations pour contrôler l'exécution d'un composant (arrêt, reprise, consultation de l'état) afin de mettre celui-ci dans un état stable permettant la modification de son contenu. Les interfaces de contrôle ne sont pas limitées au set standard, il est possible d'en créer de nouvelles. Dans sa partie interne, le contrôleur déclare un ensemble d'interfaces fonctionnelles *client* et *serveur* (internes) et des interfaces de contrôle internes. Ces interfaces internes expriment les liens entre le contrôleur et le contenu. Un contrôleur est chargé de mettre en œuvre une politique de gestion des aspects non-fonctionnels du composant, par interception des invocations à ses opérations définies dans ses interfaces fonctionnelles *serveur* externes et par interception de ses invocations définies dans ses interfaces fonctionnelles *serveur* internes. Cette politique est configurée en fonction des appels aux opérations définies dans ses interfaces de contrôle externes.

Le contenu d'un composant Fractal est un graphe d'instances de composants inter-reliés par des liens d'assemblage (*bindings* en anglais) qui mettent en relation les interfaces *client* avec des interfaces *serveur* compatibles. Le contenu est une sous-architecture qui met en œuvre les fonctionnalités métier du composant. Il existe plusieurs sortes de *bindings*. Un *normal binding* lie une interface *client* à une interface *serveur* de sous-composants. Un *export binding* lie une interface *client* interne d'un composant à une interface *serveur* externe d'un de ses sous-composants. A l'inverse un *import binding* lie une interface *serveur* interne d'un composant à une interface *client* externe d'un sous-composant.

La spécification de Fractal propose un classement des composants en différentes catégories : un composant exposant son contenu est appelé *composite* ; un composant n'exposant pas son contenu mais déclarant au moins une interface de contrôle est appelé *primitive component* ; un composant sans aucune interface de contrôle ni contenu est appelé *base component*. Les *base components* et les *primitive components* servent de condition d'arrêt à la récursion dans une architecture.

Object Web propose une première implantation de Fractal nommée Julia. Julia est un framework de classes Java qui peuvent être étendues afin de créer les différents composants d'une architecture logicielle. Un environnement graphique a également été développé afin de donner aux utilisateurs les moyens de manipuler facilement la syntaxe graphique de Fractal en vue d'une génération de code dans le framework Julia.

Fractal est, selon nous, une contribution majeure pour le paradigme du développement par composants. Le modèle abstrait de Fractal repose sur une spécification "souple", définie à partir de concepts simples, intuitifs et indépendants de toute solution technologique. Le concept de contrôleur, dérivé mais différent de celui de conteneur, permet de gérer dynamiquement les aspects non-fonctionnels des composants. Le concept de composite, facilite le travail des développeurs, en leur permettant de décrire une architecture à différents niveaux d'agrégation. Tout ceci fait de Fractal un langage adéquat pour "discuter" d'une architecture à base de composants. Néanmoins, le modèle Fractal n'intègre pas la notion de port, ce qui rend la définition du terme *interface* assez délicate. Une interface y est à la fois considérée comme un point de connexion et comme un contrat fonctionnel. Les concepts de port et d'interface nous semblent intuitivement différents et complémentaires et mériteraient d'être différenciés dans un modèle "universel" comme Fractal. Fractal est également limité par ses propres postulats : le choix de la méthode de gestion du code non fonctionnel, est laissé aux plates-formes implantant le modèle Fractal. Puisqu'il n'existe pas de spécification pour la gestion des aspects non-fonctionnels, la réutilisation de composants à travers différentes implantations du modèle semble difficile, voire impossible.

4.2.6 Cooperative Objects

Le modèle des objets coopératifs [16] (CoOperative Objects en anglais) a été proposé par l'IRIT (Institut de Recherche en Informatique de Toulouse) en 1992. Les objets coopératifs sont dédiés à la construction d'applications fortement distribuées dans lesquelles une gestion de la concurrence [101] est primordiale. Le modèle des objets coopératifs est issu d'une "fusion" des réseaux de Petri à objets (RdPO) [100] (formalisme également proposé par l'IRIT) et de la programmation par objets. Les objets coopératifs tirent parti du formalisme des RdPO, des langages objets et dans une moindre mesure des intergiciels à composants, afin de coupler leurs atouts respectifs.

Dans le modèle des objets coopératifs existent les concepts de classes (dites CoOperative Object Class, COO Class) et d'objets instances de ces classes. Les classes sont décrites classiquement suivant deux parties : leur spécification et leur implantation. la figure 4.7 présente un exemple de classe coopérative représentant un philosophe participant au "fameux" dîner et qui montre un exemple de ces deux parties. La spécification d'une classe Coopérative contient la définition de l'interface de la classe. Elle contient la déclaration d'un ensemble d'attributs et d'opérations synchrones (au minimum

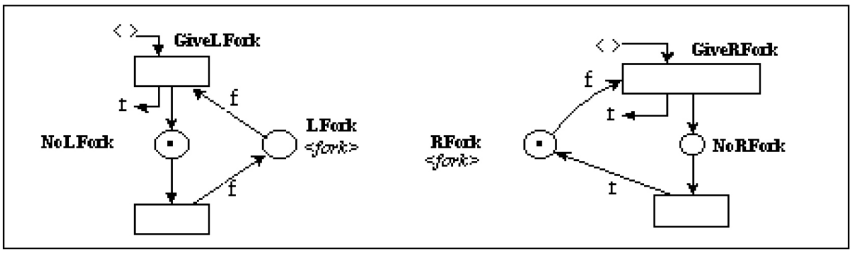
une opération d'initialisation). Elle contient la description d'un ensemble de services (opérations asynchrones), chaque service étant décrit par un nom, une liste de paramètres d'entrée et une liste de paramètres de retour. Un exemple de service est `GiveLFork`, il représente le fait qu'un client d'un objet `SPhilo` va demander à son voisin de lui donner la fourchette qu'il tient dans la main gauche. Enfin, l'interface contient une OBCS (pour *OBject Control Structure*) qui modélise le comportement concurrent des instances de la classe. Le formalisme utilisé pour décrire une OBCS est une extension du formalisme des RdPO. Chaque service défini au niveau de l'interface, correspond à une transition dans l'OBCS. Une telle transition, appelée transition de service, est mise en évidence par un arc d'entrée et un arc de sortie qui ne sont reliés à aucune place et qui représentent les points d'entrée et de sortie des paramètres associés au service (cf. service `GiveLfork`, fig. 4.7). L'OBCS peut également contenir des transitions dites internes, qui ne sont directement associées à aucun service et qui représentent des changements d'états internes à un objet coopératif. L'OBCS décrit une forme de contrat sur l'utilisation des services en fonction d'un état abstrait de l'objet (l'état étant caractérisé par le marquage des places, qui définit quelles transitions sont tirables). Elle décrit des préconditions et des postconditions (au niveau des transitions de service) associées à l'utilisation des services d'un objet coopératif, à l'image de ce qui est proposé dans le langage Eiffel.

La partie implantation d'une classe coopérative contient :
- Un ensemble d'attributs privés, qui sont propres à l'implantation.
- Un ensemble d'opérations privées classiques en POO.
- Une OBCS d'implémentation. L'OBCS d'implantation comporte plus d'informations que l'OBCS de spécification, à tel point que l'OBCS peut à ce niveau être vue comme du code (écrit dans un langage de plus haut niveau que les langages de programmation objets classiques). A chaque place est associé l'ensemble des typologies (décrites par des tuples de classes) des jetons admis dans la place. A chaque transition de l'OBCS d'implantation, sont associés des appels aux opérations internes ou aux attributs de la classe. Les transitions peuvent également contenir des appels aux services ou aux opérations, des demandes de lecture des attributs, et/ou des instanciations, d'autres objets coopératifs. Le code associé à chaque opération interne et à chaque transition peut être écrit dans un langage de programmation procédural quelconque (les auteurs ont fait initialement le choix du C++).

L'activité d'un objet coopératif consiste à exécuter son OBCS comme tâche de fond via un joueur de RdPO. Un joueur de RdPO est associé à chaque objet coopératif, dont il interprète le RdPO. Un objet coopératif correspond, à l'exécution, à un processus système. Les objets coopératifs interagissent autour d'un protocole client/serveur, basé sur un mécanisme de communication asynchrone : l'attente d'un retour de service ne bloque pas forcément l'intégralité de son comportement, qui peut par ailleurs évoluer sur d'autres parties de son OBCS. Un système d'objet coopératifs peut donc être vu comme un ensemble de processus communicants. L'environnement Syroco [102] est la seule plate-forme, à notre connaissance, implantant (en C++) le modèle des objets coopératifs. Il est basé sur l'intergiciel CORBA pour ce qui est de la gestion l'interopérabilité.

Le modèle des objets coopératifs permet la description formelle du comportement de systèmes concurrents. Cette description se fait sous la forme d'un RdPO global qui représente le comportement coordonné de tous les objets du système. Ce RdPO global est déduit de la composition des RdPO des objets coopératifs constituant le système. Cette notation formelle permet, via des outils logiciels, d'analyser des RdPO afin de prouver la validité de certaines propriétés du système (e.g. respect d'invariants, pas d'inter-blocages, etc.). Des procédures de validation formelles permettent également de savoir si l'OBCS d'implantation d'une classe est conforme à l'OBCS de son interface. Le formalisme des objets coopératifs a été appliqué à la spécification comportementale d'objets distribués CORBA [113] [17]. Le principal apport de cette proposition est de permettre la description d'une interface IDL de la même façon qu'est décrite la partie spécification d'une classe d'objets coopératifs. Le langage CORBA-IDL permet de décrire les différents services de l'objet distribué CORBA alors qu'un RdPO

```
class SPhilo specification;
operations
 Init (n: OCNAME, l: SPhilo*, r: SPhilo*, f: fork) ///
  setname (n);               //assign a unique name to the Object
  ln = l; rn = r; nbeating = 0;
  NoLFork.ADDTOKEN (NoLFork.MakeToken ());
  RFork.ADDTOKEN (RFork.MakeToken (f));
  ///;
Services
 GiveLFork (): <fork>;
 GiveRFork (): <fork>;
OBCS                  //for documentation purpose only
```




```
end.

class SPhilo implementation;
Attributes
 rn : SPhilo*;              //the right and left neighbours
 ln : SPhilo*;
 nbeating: short;           //counts how many times the Sphilo
eats
Operations
 eat() ///
   nbeating - nbeating + 1; ///;
OBCS
```

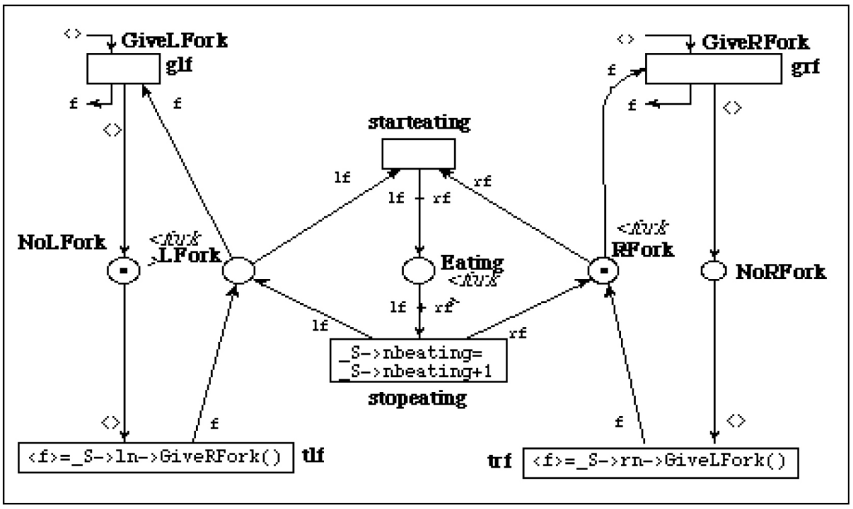


FIG. 4.7 – Exemple d'une classe d'objet coopératif [102]

permet de décrire son comportement concurrent (son OBCS). Cette proposition a permis de mettre en exergue la nécessité d'ajouter des informations sur le comportement des objets distribués dans un langage formel, afin de faciliter et de fiabiliser leur intégration [18].

Le modèle des objets coopératifs n'est pas considéré dans la littérature, comme un modèle de composant (les concepts d'interface requise ou de port requis n'existent pas de façon explicite), mais comme une extension des langages orientés objets, dédiée au développement de systèmes distribués et concurrents. Pourtant ce modèle intègre explicitement certaines des spécificités des modèles de composants : séparation de l'interface du composant (la partie spécification de l'objet) de son implémentation (le code et l'OBCS), exécution dans un monde distribué dans lequel apparait la concurrence. Sur ce dernier sujet, l'apport de ce modèle est l'utilisation d'un formalisme graphique (les réseaux de Petri à Objets) qui a une utilité multiple. Premièrement, il sert à décrire le contrat comportemental que respecte un objet coopératif, permettant de réduire les erreurs d'utilisation de cet objet. Deuxièmement, il est associé à des procédures d'analyse qui permettent de valider une composition d'objets coopératifs. Troisièmement, il sert de langage de programmation (de l'OBCS d'implémentation) de haut niveau, remplaçant un langage procédural classique, peu adapté à une description de la concurrence. Enfin, le code RdPO est exécutable par un joueur de RdPO, ce qui permet d'éviter des erreurs dues au codage du RdPO sous forme procédurale et permet donc de certifier à l'exécution, les résultats issus de la phase d'analyse.

4.2.7 Rapide

Rapide [118] est un ADL proposé conjointement par l'université de Standford et l'industriel TRW inc. L'objectif principal de Rapide est de fournir les formalismes et outils nécessaires à la vérification d'architectures logicielles grâce à un environnement de simulation. Rapide repose sur les concepts de type de composant (correspondant ici à celle d'interface), d'architecture (équivalent au concept de configuration) et de module (assimilable à une fabrique d'instances d'un type de composant donné). Un composant est une instance d'un type de composant qui a une des formes suivantes :

- Il est uniquement défini par son type, c'est-à-dire qu'il ne contient pas d'autres informations que celles contenues dans son interface. Il est alors vu comme un composant abstrait.
- Il est défini par son type et par un module. Il est alors vu comme un composant primitif.
- Il est défini par son type et par une architecture. Il est alors vu comme un composite, qui agrège une architecture. Grâce à cette propriété, Rapide permet la description d'une architecture à différents niveaux d'agrégation.

Chaque type de composant (cf. fig. 4.8) est défini par une interface déclarant un ensemble de propriétés communes à tous les composants instances de ce type. Une interface déclare des points d'échanges de messages synchrones, appelés services, qu'un composant fournit (services *Provides*) et requiert (services *Requires*). Elle contient la déclaration de points d'échanges de messages asynchrones appelés *Actions*. Une *Action* correspond soit à un point réception de messages (action *In*), soit à un point d'émission de messages (action *Out*). Toute communication entre composants est considérée comme un échange de messages (cas synchrone ou asynchrone). Rapide repose sur une notation formelle, basé sur le modèle *posets* (Partially Ordered Set of EvenTs) [79], qui permet de décrire dans une interface, le comportement observable d'un composant (i.e. l'ordonnancement de ses messages reçus et émis). Un comportement observable est décrit via un *patron d'événements* qui définit les règles de dépendances sur les messages reçus et émis par un composant. Un *patron d'événement* est défini à l'aide d'opérateurs comme celui de dépendance causale (`->`), celui d'indépendance (`||`) ou encore celui de simultanéité (`and`). Il est utilisé pour décrire une restriction imposée au comportement d'un type de composant, quant à l'ordre d'échange des messages lors de communications et quant à la nature des données échangées.

```
type Producer( Max: Positive) is interface
  action out Send(N: Integer);
  action in Reply(N: Integer);
behavior
  Start => Send(0);
  (?X in Integer) Reply(?X) where ?X<Max => Send(?X+1);
end Producer;

type Consumer is interface
  action in Receive(N: Integer);
  action out Ack(N: Integer);
behavior
  (?X in Integer) Receive(?X) => Ack(?X);
end Consumer;

architecture ProdCon() return SomeType is
  Prod: Producer(100);
  Cons: Consumer;
connect
  (?n in Integer)
  Prod.Send(?n) => Cons.Receive(?n);
  Cons.Ack(?n) => Prod.Reply(?n);
constraint
  observe from Prod.Send, Cons.Ack
    match ((?X:Integer)(Prod.Send(?X) → Cons.Ack(?X)))↑(→*);
  end;
end architecture ProdCon;
```

FIG. 4.8 – Exemple de types de composant et d'architectures dans Rapide, tiré de [118]

Les modules sont des procédures de génération d'instances de composants. Un module prend en paramètre un ensemble de composants et retourne un composant. Chaque module contient l'ensemble des fonctions implantant les services et actions contenus dans un type de composant, ainsi qu'un ensemble d'attributs qui sont des des instances de composants. Lorsqu'un module est exécuté, il génère une instance de composant, dont le type correspond à celui que le module retourne. Il contient une fonction spécifique décrivant le comportement global du composant généré. Les fonctions et le comportement global sont écrits dans un langage procédural. Ils font référence aux services fournis et requis, aux actions in et out décrites au niveau de l'interface du composant, ainsi qu'aux attributs définis au niveau du module. Le code contenu dans un module n'est utile que pour générer du code exécutable sur un simulateur spécifique.

Les architectures sont des modules spécifiques (cf. fig. 4.8). Elles décrivent un graphe d'instances de composants. Elles contiennent la définition d'un ensemble de types de composants et leurs différentes instances, déclarées comme des attributs. Une notation du langage est utilisée pour décrire les règles d'interconnexions, en terme de compositions de services requis et fournis et de compositions d'actions in et out. Ces règles sont appliquées aux attributs d'une architecture, c'est-à-dire aux composants qu'elle contient. Une règle d'interconnexion est décrite par une expression contenant un patron d'événement à vérifier (à gauche), un patron d'événement à déclencher en cas de succès de la vérification (à droite) et un opérateur. Il existe trois types d'opérateurs de connexion. L'opérateur `To` qui permet de mettre en relation uniquement deux composants (un émetteur dans le patron à gauche et un récepteur dans le patron à droite de l'opérateur). Dans le cas où le patron d'événement à gauche de l'opérateur est vérifié sur l'émetteur, le patron d'événement à sa droite déclenche la réaction associée sur le récepteur. L'opérateur |>| connecte deux patrons impliquant n'importe quel nombre de récepteurs. Si la partie gauche de l'expression est vérifiée alors tous les composants concernés par la partie droite reçoivent les messages définis dans le patron. Enfin l'opérateur `=>` ajoute la notion

d'ordre d'évaluation aux opérations précédentes.

Rapide permet la description formelle du comportement "attendu" des composants au sein de leurs interfaces. Il est fourni avec un environnement logiciel de simulation qui se nourrit des informations contenues dans les interfaces afin d'analyser, par simulation, la validité des implantations des composants (définies dans les modules). Cet environnement analyse la conformité, à l'exécution, entre un comportement décrit au niveau d'un module et les contraintes exprimées par des *patrons d'événements* au niveau de l'interface d'un composant (Le module étant une implantation du type d'un composant). La description à différents niveaux d'agrégation est permise grâce à la notion d'architecture comme implantation et fabrique d'un type de composant. Lors de la définition d'une architecture, les sous-architectures sont alors cachées aux yeux du concepteur qui ne se concentre que sur un niveau d'agrégation (à la différence de Fractal par exemple). Rapide laisse de côté bon nombre de préoccupations. Les interactions réifiées à travers des connecteurs n'y apparaîssent pas. Des préoccupations comme l'utilisation de services non-fonctionnels, le déploiement (e.g. prise en charge des contraintes de déploiement) et l'exécution, ne sont également pas prises en compte.

4.2.8 Wright

Wright [6] est un ADL développé à l'Université Canergie Mellon de Pittsburg. Sa préoccupation principale est la description formelle d'architectures logicielles. Wright regroupe toutes les caractéristiques principales d'un ADL, c'est-à-dire les entités de première classe que sont les composants, les connecteurs et les configurations.

Un composant est une entité de traitement abstraite (qui n'a pas d'implantation associée). Il possède un type qui définit un ensemble de ports. Chaque port représentant un point d'interaction du composant avec le reste de son environnement. Chaque port est associé à une description formelle qui décrit son comportement associé. Un type de composant contient également la description formelle d'un comportement global (`Computation`, fig. 4.9). Toutes les descriptions comportementales sont exprimées via le formalisme CSP (Client Server Protocol). A chaque port et chaque type est donc potentiellement associée une expression écrite en CSP. Notons que, comme dans Rapide, les auteurs définissent le concept d'interface comme équivalent à celui de type de composant, le concept d'interface n'est donc pas rattaché à celui de port.

Wright permet de spécifier des interactions entre composants via des *connecteurs*. Comme un composant, un connecteur est défini via un type décrit par une interface. Un connecteur est déclaré de manière explicite dans le langage, indépendamment des composants. Il décrit un protocole d'interaction entre des rôles jouables par des composants. L'interface d'un connecteur décrit les différents rôles joués dans l'interaction, ainsi que la description formelle du comportement associé à chaque rôle. L'interface d'un connecteur contient également la description de la "glue" (`glue`, fig 4.9) qui lie les différents rôles de l'interaction. Cette "glue" correspond à un ensemble de synchronisations et d'échanges de messages entre rôles.

La troisième sorte d'entité manipulable dans Wright est la *configuration*, qui permet de décrire des architectures. Une configuration contient la définition d'un ensemble de types de composants et de types de connecteurs, ainsi qu'un graphe biparti qui décrit la façon dont sont reliées les instances de ces types. Ce graphe est décrit à partir d'"attachements" entre les rôles des connecteurs et les ports des composants. Lorsqu'un port est attaché à un rôle, cela signifie que le port doit suivre les règles qu'énonce ce rôle afin d'être considéré comme "légal" dans l'interaction définie par le connecteur. Les configurations sont particulièrement utiles à la description d'architectures à différents niveaux d'agrégation, en permettant d'agréger des compositions de composants. Une configuration possède une interface qui définit son type. Un port défini au niveau de l'interface d'une configuration peut être "attaché" à un port d'un composant qu'elle contient. Grâce à ce mécanisme, les configurations

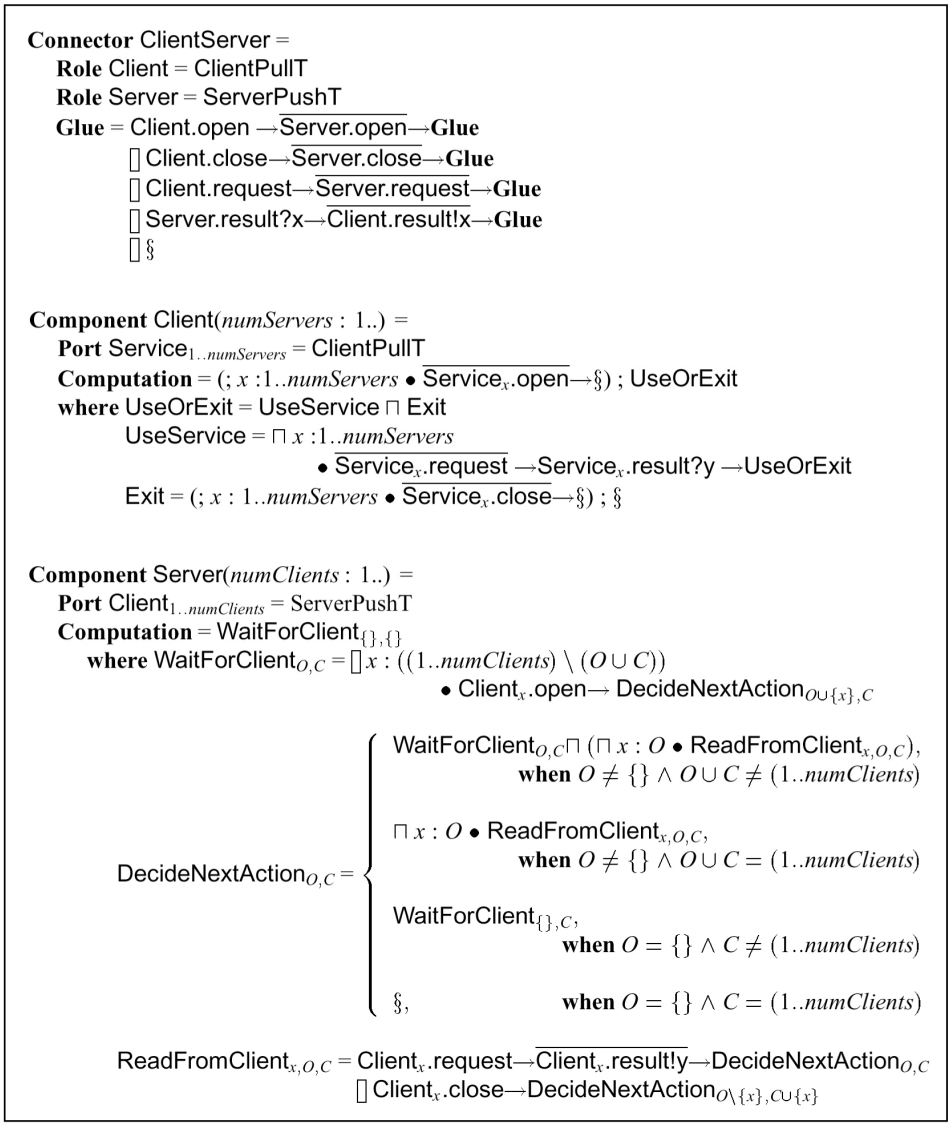


FIG. 4.9 – Exemple de composants et connecteurs dans Wright [6]

sont donc considérées comme des composants partoculiers, ce qui permet de les composer avec des composants ou des configurations.

L'idée essentielle développée dans Wright est de décrire différents aspects d'une architecture via différentes sortes d'entités manipulables, tels que les connecteurs et les configurations. Ces différentes catégories sont définies en fonction des préoccupations qu'elles permettent de décrire : une architecture via une configuration, un protocole d'interaction via un connecteur, une unité de traitement et de stockage de données via un composant. L'expression des comportements formels associés à chaque port, rôle et interface donne à Wright un pouvoir d'expression important. Cela permet de modéliser des applications distribuées dans lesquelles ont lieu des interactions complexes. Des outils logiciels dédiés permettent d'effectuer des analyses sur une spécification CSP : consistence entre comportement d'un composant et comportement de ses ports, connecteur et rôles sans inter-blocages, compatibilité d'un attachement entre un port et un rôle. Un reproche que l'on peut faire à Wright, est qu'il n'est supporté, à notre connaissance, par aucun environnement logiciel qui permette d'intégrer des implantations de composants logiciels et de tester la conformité du résultat de leur composition par rapport à une description architecturale. D'autre part, la notation formelle CSP nous semble trop complexe à manipuler pour être utilisée, telle quelle, par une large gamme de développeurs.

4.2.9 MétaH

MétaH [122] est un ADL proposé par l'institut de Technologie Honeywell depuis 1996. Il est dédié à la description d'architectures logicielles dans le cadre des applications de guidage, de navivation et de contrôle [24]. MetaH est vu comme un ADL formel, puisqu'il propose un ensemble de notations permettant de spécifier différentes propriétés des composants. Il est supporté par un ensemble d'outils logiciels, utilisant ces spécifications pour proposer aux développeurs des procédures d'analyse/validation d'une architecture. MetaH a aussi la particularité de générer le code équivalent à l'architecture via l'intégration de code source Ada.

Dans MetaH un composant est appelé *Process* (fig. 4.10). Il est défini comme une unité exécutable de traitement d'information. C'est une unité d'ordonnancement qui possède son propre flot d'exécution et sa propre adresse mémoire. Il est décrit à partir d'un type et d'un ensemble d'implantations. Son type est défini via une interface qui liste l'ensemble de ses propriétés visibles. Ces propriétés sont les ports, les événements, les packages et les moniteurs. Les ports sont des points d'échange de messages. Ils sont *in* s'ils sont des points d'entrée de messages ou *out* s'ils sont des points de sortie de messages. Ils sont typés en fonction d'un type défini au niveau d'un package (cf. plus loin). Un événement est un identifiant d'un signal système, que le *Process* est suceptible de lever ou recevoir lors de son exécution. Un package est une collection de sous-programmes et de déclarations d'objets statiques, faites dans le langage de programmation Ada. Un package est conforme à un type de package qui contient un ensemble de types (en Ada), utilisés pour typer les ports d'un *Process*. Un package peut être partagé de façon non exclusive entre plusieurs *Process*. Un moniteur est un package spécifique, qui peut être partagé de façon exclusive entre plusieurs *Process*. Une implantation d'un *Process* sert uniquement à fixer un ensemble de valeurs d'initialisation comme la fréquence du *process* à l'exécution ou la location du code source utilisé. Un *Process* peut être périodique (ce qui signifie que le processus correspondant est déclenché à des intervalles de temps réguliers) ou apériodique (s'il est déclenché uniquement sur arrivée d'un événement).

MetaH propose plusieurs catégories d'entités assimilables à des configurations (i.e. composants composites) : les *Macros* et les *Modes*. Les *Macros* sont des entités qui contiennent la description d'une collection de *Process* et de connexions établies entre ces *Process*. Les *Modes* sont des *Macros* spécifiques. A un moment donné et pour une *Application* donnée (cf. plus loin), un seul *Mode* est actif, à l'inverse des *Macros* qui s'exécutent en parallèle. Le type d'une *Macro* ou d'un *Mode* (i.e. son

```
with DOMAIN_TYPES;
package P1_PORT is
   TO_P2 : DOMAIN_TYPES.INTEGER_TYPE;
   FROM_P2: DOMAIN_TYPES.INTEGER_TYPE;
end P1_PORT;

with MetaH;
with P1_PORT;
procedure P1 is
begin
   P1_PORT.TO_P2.FRESH := TRUE;
   P1_PORT.TO_P2.DATA := 0;
   loop
      MetaH.Await_Dispatch (TRUE);
      if P1_PORT.FROM_P2.FRESH then
         P1_PORT.TO_P2.DATA := P1_PORT.FROM_P2.DATA + 1;
      end if;
   end loop;
end;

macro M is
end M;

macro implementation M.SIMPLE is
   P1: periodic process P1.SIMPLE;
   P2: periodic process P2.SIMPLE;
connections
   P1.FROM_P2 <- P2.TO_P1;
   P2.FROM_P1 <- P1.TO_P2;
end M.SIMPLE;

application SIMPLE is
   macro M.SIMPLE on processor I80960MC.CVME_TIMED;
attributes
   I80960MC'ClockPeriod := 500 ms;
end SIMPLE;
```

FIG. 4.10 – Exemple de process et de mode dans MetaH, tiré de [122]

interface), contient la description d'un ensemble de ports, d'événements, de packages et de moniteurs, comme les *Process*. Les connexions entre ports des *Process*, décrites au niveau d'un *Mode* ou d'une *Macro*, doivent respecter la compatibilité des ports. La compatibilité entre les ports est fonction de leur statut (e.g. un port *in* assemblé avec *out*) et de leur types. Les connexions entre événements sont utilisées pour connecter un événement *out* soit avec un événement *in*, soit avec un *Process* apériodique (activation du *Process*), soit avec un *Mode* (activation du *Mode*). Le partage de packages et de moniteurs entre *Process* est décrit à l'intérieur d'un *Mode*, à l'aide de connecteurs spécifiques.

MétaH propose des notations permettant de représenter l'infrastructure matérielle d'exécution d'une architecture logicielle. Ces notations sont basées sur des entités qui représentent chacune des composants matériels.

- Les *Processors* sont des composants matériels capables d'exécuter du code, dont le type définit un ensemble de ports.
- Les *Memories* sont des entités matérielles de stockage de mémoire qui peuvent être partagés entre des *Processors*.
- Les *Devices* sont des composants matériels qui ne peuvent pas exécuter du code logiciel, mais qui peuvent partager de la mémoire avec des *Processors*. Leur type définit un ensemble de ports.
- Les *Channels* sont les entités matérielles qui connectent les ports des *Processors* et des *Devices* (Bus, mémoire partagée).
- Les *Systems* sont les entités matérielles qui permettent de définir des groupes d'éléments matériels connectés entre eux. Une connexion permet de décrire le partage de *Memories* entre des *Processors*. Une connexion permet également de décrire la façon dont les *Processors* et les *Devices* sont reliées par des *Channels*. Un *System* est une représentation de l'infrastructure matérielle d'une application de contrôle.

La dernière catégorie d'entités proposée est celle des *Applications*. Une *Application* est l'entité matérielle/logicielle qui agrège la description complète d'une aplication de contrôle. Elle contient la description des *Macros* et *Modes* constituant une architecture logicielle, ainsi qu'un *System* qui représente l'infrastructure matérielle sur laquelle l'architecture logicielle est déployée. Au niveau d'une *Application*, les ports, les événements, les packages et les moniteurs des composants matériels comme des composants logiciels peuvent être connectés, ce qui permet de décrire l'interaction entre le logiciel et le matériel.

En plus de cette description matérielle/logicielle, MetaH fournit un ensemble de notations s'appliquant aux différentes catégories de composants proposés et à partir desquelles peuvent être réalisées deux sortes d'analyses : l'analyse de l'ordonnancement des *Process* et l'analyse de la fiabilité des *Applications*. Les procédures d'analyse sont basées sur des propriétés non-fonctionnelles, appelées *Attributes*. Les différents *Attributes* utilisables font partie d'un vocabulaire spécifique à chaque procédure d'analyse et ont été établis de manière empirique. Ces *Attributes* sont valués par l'utilisateur, en fonction de leur nature. Certains *Attributes* sont, en particulier, valués par des expressions appelées *Paths*. Un *Path* permet de représenter un comportement séquentiel d'un composant (i.e. *Process*, *Macro*, *Mode*), en fonction des informations contenues dans les différentes entités proposées par MetaH (e.g. événements, messages, moniteur, etc). L'analyse de l'ordonnancement permet de vérifier et/ou observer : la validité de l'ordonnancement des *Process*, le temps moyen de processeur utilisé par chaque *Process*, des informations sur leur période d'exécution (e.g. respect d'échéances, etc.). L'analyse de la fiabilité permet de déterminer la probabilité de pannes et la tolérance aux fautes d'une *Application* sujette à l'arrivée d'erreurs aléatoires. Cette analyse se base sur des *Attributes* spécifiques, valués avec des modèles d'erreur. Ces modèles d'erreur permettent de spécifier les types de fautes qui peuvent survenir aléatoirement, les états d'erreur dans lesquels les composants logiciel et entités matérielles (*Process*, *Macros*, *Modes*, *System*, *Application*, etc.) peuvent se trouver lors d'une faute, ainsi que la réaction des composants logiciels à l'occurrence d'une faute (prise en charge

ou propagation).

MetaH a été conçu avec une vision pragmatique du développement de contrôleurs fiables. Cet ADL n'est pas basé sur une notation formelle, comme Rapide ou Wright. Sa notation est entièrement construite à partir d'un ensemble de propriétés définies de façon empirique. L'analyse (de fiabilité ou d'ordonnancement) d'une description d'architecture est réalisée par des outils logiciels performants et spécialisés. Le pouvoir d'expression de MetaH reste néanmoins relativement limité comparé à d'autres ADLs, puisque les *Attributes* sont prédéfinis et de nouveaux attributs ne peuvent pas être ajoutés par un utilisateur. Les préoccupations gérées par MetaH sont innovantes, cet ADL adressant spécifiquement les problèmes d'ordonnancement et de fiabilité de systèmes temps-réel embarqués. Afin de gérer complètement ces problèmes, MetaH propose de représenter les éléments matériels via des entités de description, au même titre que les composants logiciels. Certaines de ces entités possèdent les mêmes moyens de composition (e.g. ports, événements) que les composants logiciels, ce qui permet de représenter les interactions entre les entités matérielles et les composants logiciels. Cette description matérielle et logicielle convient particulièrement au développement de contrôleurs, permettant par exemple d'adapter les procédures d'analyse et de génération de code en fonction de l'infrastructure matérielle embarquée. MetaH est, en effet, un des rares ADL qui permette de générer le code exécutable d'une application, par le biais d'outils spécifiques qui se basent sur une description architecturale (pour générer le code squelette en Ada) et qui permettent d'intégrer des briques logicielles (écrites à partir de modules Ada) implantant soit les composants logiciels de l'*Application*, soit le code d'interfaçage avec les éléments matériels de l'*Application*. Cet outillage fait de MetaH un des ADL les plus directement utilisables dans un contexte industriel.

4.2.10 ArchJava

ArchJava [4] est une extension du langage Java qui permet aux programmeurs de décrire une architecture logicielle grâce à du code Java. ArchJava peut donc être vu comme une proposition visant à intégrer les notions de haut niveau, propres aux architectures logicielles, à l'intérieur d'un langage de programmation orienté objet. De la même manière que les autres ADL présentés, ArchJava propose une syntaxe pour décrire une architecture logicielle, mais la différence notable est que cette syntaxe est directement exprimée à partir de structures ajoutées au langage de programmation Java.

ArchJava permet de manipuler les concepts de *composant* et de *classe de composant* (fig. 4.11). Un composant est une instance d'une classe de composants. Il communique avec d'autres composants via ses *ports*. Un port est un point de connexion servant de canal de communication avec un ou plusieurs composants. Un port est instance d'un *type de port*, qui définit une combinaison de signatures de méthodes *fournies* et *requises*. Un type de port est défini au niveau d'une classe de composants, qui contient l'implantation de chaque méthode *fournie* déclarée dans un de ses types de port. Une méthode d'implantation d'une classe de composant peut contenir, dans son corps, des appels à n'importe quelle méthode *requise* déclarée dans un de ses types de port. Un composant peut posséder plusieurs ports instances d'un même type de port (ce type étant appelé *interface de port*).

La notion d'architecture (i.e. configuration) existe à travers des composants composites : ce sont des composants qui sont eux-mêmes construits à partir d'une interconnexion de composants, alors appelés sous-composants. Les sous-composants d'un composite sont des attributs déclarés au niveau d'une classe de composants. Les connexions autorisées entre ports des sous-composants sont exprimées via des *connect patterns*. Un *connect pattern* permet d'exprimer la possibilité d'une connexion entre les types de ports concernés et pour les classes de composants concernées (celle des sous-composants connectés) (cf. fig. 4.11). Les connexions en elles-mêmes sont établies au moment de l'instanciation d'une classe de composant composite, via l'appel d'une directive *connect* qui met en relation un ensemble de ports appartenant à différents sous-composants. Cette directive ne créera de connexion

```
public component class Parser {
      public port in {
             requires Token nextToken();
      }
      public port out {
              provides AST parse();
      }
      AST parse() {
              Token tok=in.nextToken();
              return parseExpr(tok);
      }
      AST parseExpr(Token tok)
      { ... }

      ...

}
public component class Compiler {
      private final Scanner scanner = new Scanner() ;
      private final Parser parser = new Parser() ;
      private final CodeGen codegen = new CodeGen() ;
      private final Assembler asm = new Assembler() ;

      connect pattern Scanner.out, Parser.in ;
      connect pattern Parser.out, CodeGen.in ;
      connect pattern CodeGen.client, Assembler.server ;

      public Compiler() {
             TCPConnector.registerObject(asm, ASSEMBLER_PORT, "server") ;

             connect (scanner.out, parser.in) ;
             connect (parser.out, codegen.in) ;
             connect (codegen.client, asm.server) ;
      }

      connect pattern CodeGen.client, Assembler.server with TCPConnector{
             client (CodeGen generator, InetAddress address) {
                     connect (generator.client, asm.server) with
                           new TCPConnector (address, ASSEMBLER_PORT,
                                 "server") ;
             }
      } ;
}
```

FIG. 4.11 – Exemple de code ArchJava, tiré de [89]

que si un *connect pattern* l'autorisant a été déclaré dans la *classe de composants*. Quand une connexion est établie entre deux ports, les méthodes fournies par un port sont liées aux méthodes requises part l'autre, ce qui permet de faire communiquer les deux composants considérés : celui dont la méthode est requise fait appel à la méthode fournie de l'autre. L'environnement d'ArchJava possède un mécanisme de vérification de type standard qui s'applique chaque fois que des ports sont connectés. Celui-ci vérifie en particulier la correspondance entre les signatures des méthodes fournies et requises des ports connectés. Par exemple, si une méthode requise par un port n'est fournie par aucun autre port connecté, c'est que la connexion est invalide. Pour les types de ports pouvant être instanciés plusieurs fois pendant la durée de vie d'un composant, les *connect patterns* sont définis lors de l'initialisation d'un composite (dans le constructeur) et non de façon statique. Chacun de ces *connect patterns* est alors associé à un *constructeur de connexions* (bloc de code Java associé au *connect pattern*, fig. 4.11), qui va exprimer la manière dont les connexions entre ce type de port et un ou plusieurs autre types de ports sont réalisées pendant l'exécution.

ArchJava propose également de manipuler des entités nommées *Connecteurs* [5]. Lorsqu'une connexion est établie entre ports de composants, une vérification de type à lieu, mais cette vérification de type est générique à l'environnement ArchJava. Or souvent, un développeur voudra programmer des contraintes spécifiques pour une connexion. Un connecteur est une entité qui va servir à vérifier la validité d'une connexion en mettant en place un mécanisme de vérification de type spécifique. Par exemple, pour le `TCPConnector` de la figure 4.11, celui-ci encapsule un mécanisme de vérification de type qui assure que tous les paramètres des méthodes, déclarées dans les ports concernés par la connexion, sont sérialisables (afin de pouvoir être échangés lors de communications réseaux). Un connecteur va également servir à encapsuler le code de communication (utilisation de TCP/IP dans l'exemple) qui met en œuvre les communications induites par la connexion.

ArchJava est une proposition innovante qui prêche pour l'intégration, dans un langage de programmation, des concepts des ADL : composants, connecteurs, configurations. Ceci amène effectivement une gestion plus aisée du code source puisque la même syntaxe est utilisée pendant le développement d'une application. Ce lien étroit entre description architecturale et langage de programmation permet de mettre en œuvre un principe que les auteurs nomment l'*intégrité des communications* : les briques logicielles d'implantation d'une application communiquent intégralement et uniquement de la façon dont les connexions sont définies dans l'architecture de cette application. Pour permettre la gestion de l'intégrité des communications, dans un monde où les composants et leurs connexions peuvent être créés dynamiquement, les auteurs proposent la notion de connect pattern (qui permet de limiter les connexions entre composants à un schéma pré-établi). De part l'orientation choisie par ses auteurs, ArchJava ne peut être considéré comme un ADL formel : il ne repose pas sur l'utilisation des notations formelles pour exprimer les comportements et contraintes des composants et connecteurs. Ceux-ci sont entièrement décrits à partir de code objet classique. Cette vision "programmatique" d'un ADL, si elle nous semble pragmatique et facile à prendre en main car basée sur un langage objet largement utilisé, manque de la réelle valeur ajoutée des ADL : leur capacité d'analyse.

4.3 Réflexions sur les composants logiciels

Nous menons dans cette section une réflexion d'ensemble sur les différentes approches de développement basées sur les composants logiciels. Nous tentons par la suite de définir les grands concepts qui caractérisent une approche générique. Nous comparons, pour chaque concept, ses définitions dans les ADL et modèles de composants, avant d'en dresser une synthèse.

4.3.1 Composant logiciel et architecture

Selon l'approche retenue (Modèle de composant ou ADL), le concept de composant n'est pas exactement le même. Dans les ADLs, un *composant* est une unité de modélisation qui contient la description des propriétés "souhaitées" d'une partie identifiée d'une application. Il est la brique de base à partir de laquelle on peut décrire une architcture. Dans les modèles de composants, il est une unité de programmation, de déploiement et d'exécution, qui encapsule des données et qui met en œuvre un ensemble de fonctionnalités. Il est la brique de base à partir de laquelle implanter une architecture. Dans les deux cas, un composant est instance d'un type, qui définit un ensemble de propriétés communes (paramétrables ou non) à un ensemble de composants.

Néanmoins, les préoccupations traitées par les ADLs et les modèles de composants se confondent en partie dans de nombreuses approches, ce qui rend parfois difficile l'identification d'une proposition comme un ADL ou un modèle de composant. ArchJava permet de décrire des architectures avec les mêmes moyens que les ADL (connecteurs, composite) mais dans un langage de programmation. MetaH est un ADL très proche des problématiques de programmation et de déploiement, puisqu'il est supporté par un outillage complet permettant l'intégration et le déploiement d'implantations dans les architectures. Le CCM permet, lui, de décrire des architectures à partir d'une notation spécifique qui décrit à la fois le déploiement et les connexions entre composants déployés. Au delà de certaines ressemblances, les deux approches semblent intuitivement complémentaires, puisqu'elles s'appliquent à des phases successives d'un processus de développement (conception, programmation, déploiement). A l'heure actuelle, les ADLs ne reposent, pour la plupart sur aucun standard ; cela les rend d'autant plus difficilement utilisables conjointement avec des modèles de composants existants. Le futur des approches à composants sera certainement fait d'approches combinant les avantages des ADLs et des modèles de composants, à l'image de Fractal, ArchJava et MetaH.

Si l'on tente de mettre en commun ces deux visions, il apparaît qu'un *composant* est une entité constitutive d'une architecture logicielle, dont la nature évolue en fonction de la phase du processus de développement considérée. Pendant la phase de conception un composant est vu comme une unité de modélisation d'une architecture logicielle. Pendant la phase de programmation un composant est vu comme une unité de code source. Pendant les phases suivantes il est vu comme une unité de déploiement et d'exécution. De la même manière, la nature d'une architecture logicielle évolue aux cours d'un processus de développement, d'une forme abstraite vers une forme exécutable. De façon optimale, composants logiciels et architectures logicielles existent pendant toutes les phases d'un processus de développement, depuis la modélisation jusqu'à l'exécution, afin d'assurer leur traçabilité et leur manipulation à tout moment de leur cycle de vie. D'une phase à une autre du processus, les descriptions des composants et des architectures sont de plus en plus détaillées afin d'arriver à un niveau de détail suffisant pour pouvoir les exécuter.

4.3.2 Interfaces, contrats et services

La définition du concept de *service* reste quelque peu évasive, voire inexistante, dans les travaux étudiés. Nous définissons un *service* comme une fonctionnalité mise en œuvre par un ou plusieurs composants, utilisable par des composants et qui se traduit par un ensemble d'échanges de messages. Un composant propose un ensemble de services, qui sont alors appelés *service fournis*. Un composant utilise un ensemble de services qui sont alors appelés *services requis*. Les concepts de services fournis et requis sont essentiel dans les approches à composants, car ils permettent d'exprimer l'offre et l'utilisation de fonctionnalités indépendamment des composants.

Les concepts de *service requis* et de *service fourni* sont de vus de façon subtilement différentes dans les approches à composants. Dans de nombreuses propositions (ArchJava, CCM, Fractal, EJB, objets coopératifs, etc.), un *service requis* d'un composant est vu comme un appel d'opération d'un

autre composant ; un *service fourni* d'un composant est vu comme une contrainte sur la présence dans son code, de l'opération implantant ce service. Dans des ADL comme Whright et Rapide, un service est vu comme un échange de messages. Un *service fourni* correspond soit à un envoi, soit à une réception de messages. Pour parler dans les termes de la programmation objet : la mise en œuvre d' un *service fourni* peut donc consister à implanter l'opération qui correspond à un message reçu ou à implanter l'appel à l'opération qui correspond à un message émis. Il en va de même pour un *service requis*. Cette approche permet de découpler l'échange de messages, des concepts de *services requis et fournis*, qui permettent d'exprimer ce que le composant sait faire pour d'autres composants et ce qu'il demande à d'autres composants. Cette idée a déjà émergé dans façon implicite dans les mécanismes de communication par enregistrement/notification d'événements, proposés dans les modèles Java Beans et CCM. Le composant qui définit une source d'événements définit deux services fournis correspondant à deux points de réception de messages (enregistrement et désenregistrement) et un service fournit correspondant à un point d'émission de messages (notification). Le puits d'événements est donc décrit par deux points de réception de messages et un point d'émission de messages ; concrètement, l'appel de l'opération de réaction à la notification est considéré comme un service fourni. A l'inverse, le puits d'événements est décrit par deux services requis correspondant à deux points d'émission de messages (demandes d'enregistrement et de désenregistrement) et par un service fourni correspondant à un point de réception de messages (réaction aux notifications). Il semble que la dissociation des *messages reçus et émis*, des concepts de *services fournis et requis* permet de définir des interactions plus complexes que de simples envois de messages.

La relation entre *interface* et *service* varie d'une proposition à une autre. Par exemple, un service est tantôt défini comme une opération (ou plus rarement un appel d'opération), tantôt comme une interface et donc par un ensemble d'opérations. Ces deux définitions sont, a priori, valides : il existe des services qui peuvent eux-mêmes être composés d'un ensemble de sous-services. Par exemple un service de gestion de compte, peut être décomposé dans un service de consultation de compte, un service de retrait d'argent et un service de dépot d'argent. Une interface contenant la déclaration d'un service ou d'un ensemble de services est elle-même composée de sous-interfaces déclarant les sous-services. Une signature de méthode ou une interface d'objet, sont des exemples de déclarations de services de différentes granularités. Le concept d'*interface* est équivalent, dans les ADL, à celui de type de composant. Ces deux concepts sont différents, et méritent d'être clairement distingués comme c'est le cas dans plusieurs modèles de composants (CCM, Fractal, Darwin, etc.). Le type de composant définit la possibilité de substituer un composant par un autre composant du même type ou d'un de ses sous-types (si la relation de sous-typage existe). Une *interface* dans les modèles de composants, sert à exprimer la façon dont est offert ou utilisé un service. Le concept d'*interface fournie* permet de décrire un *service fourni*. Un même composant peut déclarer un ensemble d'*interfaces fournies* qui correspondent à tous les *services fournis* qu'il déclare. Le concept d'*interface requise* permet de décrire un *service requis*. Un même composant peut déclarer un ensemble d'*interfaces requises* pour chaque *service requis* qu'il déclare. Pour généraliser, les *interfaces* servent à exprimer des *contrats* (offre ou utilisation) sur des *services*. La définition d'*interfaces* est le pilier de la réutilisation dans une approche basée sur les composants logiciels, puisqu'elles définissent les possibilités d'assemblage des composants. Elles permettent de définir des services fournis et requis, indépendamment de la façon dont ils ont implantés dans différents composants. Cela se révèle particulièrement utile dans un monde où plusieurs industriels proposent des composants différents, interopérant autour de services communs, qui ont potentiellement été définis à partir de standards existants. C'est une des motivations qui nous pousse à penser que le paradigme des composants est bien adapté à la satisfaction des enjeux actuels et futurs de la robotique de service.

4.3.3 Connexion, connecteurs et ports

Le principe de *connexion* entre *ports* des composants logiciels, est la base du mécanisme de composition traditionnellement appelé *assemblage*. Un *port* est un point de connexion d'un composant, qui permet de l'assembler avec d'autre composants, afin qu'ils interagissent. Un *port* est typé par une interface qui détermine quel service est offert ou utilisé via ce port. Un *port fourni* est un point de connexion à un *service fourni* par un composant. Un *port requis* est un point de connexion à un *service requis*. Une *connexion* permet de mettre en relation des *ports* afin de réaliser un assemblage de composants. L'orsqu'une *connexion* est établie entre des ports, une vérification de la compatibilité des interfaces qui les typent est réalisée, afin de déterminer si l'assemblage est valide.

Le concept de *connecteur*, issu des ADLs, est également particulièrement important. Un *connecteur* est une entité encapsulant un protocole d'interaction entre composants. C'est une *connexion* particulière, car un *connecteur* est une entité de première classe. Dans les ADLs formels, un *connecteur* est une entité de modélisation décrivant un protocole que doivent suivre les composants qu'il connecte, alors que dans une approche plus pragmatique comme ArchJava, il encapsule l'implantation d'un procotole et il est vu comme une entité exécutable à part entière. De la même façon qu'un composant, un *connecteur* peut être vu comme une entité évoluant tout au long du cyle de vie d'un logiciel, depuis la forme d'unité de modélisation d'un protocole d'interaction, jusqu'à celle d'unité de communication à l'exécution. De façon générique, un *connecteur* définit l'ensemble des messages que des composants échangent, la façon dont ils se synchronisent, les contraintes qu'ils imposent sur les *ports* mis en relation (e.g. contraintes sur les interfaces). Nous voyons une correspondance entre le concept de *connecteur* et celui de *mode de communication* associé à un *port*. Dans différentes propositions, (e.g. CCM, JavaBeans), un *port* est associé un *mode de communication* qui définit le protocole utilisé pour faire interagir les composants connectés (classiquement par envoie de requêtes simples, ou par notification d'événements). Dans ces approches, les modes de communication sont limités à un ensemble prédéterminé de protocoles, définis dans le modèle de composant. Avec le concept de *connecteur* tel qu'il est proposé dans Wright et ArchJava, les protocoles sont réifiés à travers des entités réutilisables (indépendamment des composants). Les développeurs peuvent créer de nouveaux *connecteurs* qui correspondent à de nouveaux modes de communication. Cette approche semble particulièrement intéressante, dans notre problématique, pour pallier le problème de la réutilisation des types d'interactions, parfois complexes, définis au sein des architectures de contrôle.

4.3.4 Composite et Configuration

Le concept de composant *composite* est également présent de manière récurrente dans différents ADL tels que Wright ou Archjava et modèles de composants tels que Fractal et Darwin. Ce concept est la base du mécanisme de composition appelé *agrégation* (que l'on peut comparer à la relation de composition d'UML). Un composite est un composant qui agrège un ensemble de composants, c'est-à-dire qu'il contrôle leur cycle de vie à l'exécution, par exemple leur assemblage. Le mécanisme d'agrégation a plusieurs atouts. Il permet aux programmeurs ou architectes de manipuler des assemblages de composants (i.e. une architecture logicielle) sous la forme d'une unité de modélisation (dans Wright et Fractal) ou de programmation (dans ArchJava et Fractal/Julia), réutilisable. Il permet également de décrire une architecture à différents niveaux de granularité, ce qui a tendance à faciliter la lecture globale d'une architecture.

Dans certains ADL, les entités appelées *configurations* sont des composites, mais qui sont considérées de manière spécifique par rapport aux composants (à la différence du modèle Fractal, dans lequel, par défaut, un composant est un composite). Une *configuration* est une entité de modélisation qui agrège une architecture logicielle, c'est-à-dire un assemblage de composants via des connecteurs. Une différenciation entre les concepts de composant et *configuration* est intéressante lorsque la description d'une architecture traite la problématique de son déploiement. En effet, le déploiement d'un

composant ne nécessite pas la gestion de problématiques aussi complexes que le déploiement d'une architecture logicielle. Par exemple, gérer le déploiement d'une *configuration* peut nécessiter la gestion de la distribution et de l'ordonnancement des composants qu'elle agrège, sur une infrastructure matérielle de déploiement. A l'image de ce qui est proposé dans MetaH à travers les concepts de *Mode*, *Système* ou *Application*, certaines propriétés liées au déploiement doivent être spécifiquement gérées au moment de la définition des configurations.

De la même façon qu'un composant, une *configuration* peut être vue comme une entité manipulable tout au long du cyle de vie d'un logiciel, depuis la forme d'unité de modélisation d'une architecture (exemple des ADLs) jusqu'à celle d'appplication déployée et exécutable (exemple des descripteurs de déploiement du CCM).

4.3.5 Comportement et Contraintes

Le comportement d'un composant est vu différemment suivant que l'on se place dans un ADL ou dans un modèle de composants. Dans les ADLs, le comportement "attendu" des composants est décrit via une notation empirique (MetaH) ou formelle (Wright). Cette notation est accompagnée de procédures permettant aux développeurs de vérifier que leurs spécifications sont valides, ou conformes à certaines attentes. Dans les modèles de composants, le comportement "réel" est décrit avec un langage de programmation, qui permet de définir complètement comment le composant s'exécute. Il existe des approches qui tentent subtilement de méler ces deux approches, à l'image des objets coopératifs. Le comportement "attendu" associé aux services définis dans une interface, est décrit via des réseaux de Petri à Objets. Les RdPO sont utilisés pour décrire le comportement "réel" qui sera exécuté. La description du comportement évolue donc d'une vision très abstraite (le comportement "attendu") vers une vision programmatique (le comportement exécuté) pendant le développement. De façon générique, on considère que la description des comportements des composants est susceptible d'évoluer tout au long d'un processus de développement, d'une version abstraite (e.g. des signatures de méthodes et une diagramme statecharts en UML) vers une version concrète (e.g. le code des méthodes).

Pour décrire les comportements des composants, les ADLs proposent deux approches. La première approche consiste à utiliser une notation formelle permettant de décrire la *sémantique comportementale* d'un composant ou d'un port. La sémantique comportementale est la façon dont les activités sont organisées et la façon dont les messages sont reçus et émis en conséquence. Elle décrit le comportement "nominal" d'un composant (globalement ou localement à un de ses ports). La seconde approche consiste à décrire un ensemble de *contraintes* sur un composant (globalement ou localement à ses ports). Une *contrainte* exprime une règle que doit respecter le composant quant à sa composition et à son exécution (e.g. refus de certains états d'un composant, refus de certains échanges de messages). Sémantique comportementale et contraintes permettent d'exprimer un même comportement "attendu" de façons différentes. Certaines parties d'un comportement sont plus intuitivement décrites via une sémantique comportementale (e.g. le patron global des activités du composant), alors que certaines parties sont plus intuitivement décrites par des contraintes (e.g. états d'erreurs, acceptation des messages). Une notation idéale pour la description de comportements "attendus" devrait donc intégrer ces deux approches complémentaires. Une notation pragmatique devrait intégrer une partie "formelle" pour la description des comportements "attendus" et devrait permettre l'intégration de code dans les comportements "attendus" afin de les raffiner vers des comportements "réels".

4.3.6 Déploiement, dépendances et services non-fonctionnels

Le *déploiement* est la phase qui consiste à diffuser, installer et paramétrer des composants sur les sites d'une infrastructure matérielle/système, afin de rendre une application prête à être utilisée. C'est

pendant cette phase que les *dépendances non-fonctionnelles* d'un composant doivent être satisfaites par le site sur lequel il est déployé. Un composant doit exhiber pour cela un ensemble de *dépendances non-fonctionnelles* et un site doit exhiber un ensemble de *services non-fonctionnels* qu'il offre et un ensemble de *caractéristiques techniques* qui lui sont propres. Un service *non-fonctionnel* est une fonctionnalité offerte par un système d'exploitation (services système, accès à un périphérique, etc.) ou un intergiciel (gestion des transactions, sécurité, etc.). Une *caractéristique technique* est une propriété du site qui peut être utilisée pour vérifier la validité du déploiement (type de l'OS, langages de programmation supportés, fréquence du processeur, etc.). Une *dépendance non-fonctionnelle* traduit le besoin d'un composant d'utiliser un *service non-fonctionnel* ou de vérifier une contrainte sur les *caractéristiques techniques* du site sur lequel il est déployé.

Pour exhiber ses *dépendances non-fonctionnelles*, un paquetage de composant peut donc contenir un *descripteur de déploiement* (notion que l'on retrouve dans CCM par exemple) qui définit des contraintes que le site accueillant le composant doit satisfaire. Ces contraintes touchent à la capacité mémoire, la puissance CPU, le système d'exploitation cible, la présence de compilateurs pour un langage de programmation spécifique, la présence de bibliothèques, la présence de services non-fonctionnels spécifiques, etc. Au delà de vérifier le respect de contraintes, de nombreux modèles de composants permettent de configurer l'utilisation, par un composant, des services *non-fonctionnels*, en fonction de son contexte applicatif. Le contexte applicatif est caractérisé par la finalité de l'application et par l'infrastructure de déploiement : l'intergiciel utilisé (services techniques de haut niveau disponibles, e.g. transaction, sécurité, persistence, etc.), le système d'exploitation (services système, e.g. pile protocolaire, procédure système, etc.), et le site (services "matériels", e.g. utilisation d'un périphérique spécifique sur le site du composant).

Afin de permettre la configuration la plus simple possible des *services non-fonctionnels*, le concept de *conteneur* a été inventé. Un *conteneur* est un environnement d'exécution pour un ensemble de composants, qui permet un accès et une utilisation simplifiée des services non-fonctionnels. Un conteneur est décrit à partir de notations spécifiques qui permettent de décrire la façon dont son configurés et utilisés certains services non-fonctionnels. Ce même principe apparait de façon différente dans Fractal, sous la forme d'un *contrôleur*, qui est la partie d'un composant qui gère les aspects non-fonctionnels (administration, persistance, transactions, etc.). Le concept de *contrôleur* est proche de celui de *conteneur* mais pas identique : il ne fournit pas nécessairement un même flot d'exécution pour les composants qu'il contient (dans le cas où ces composants sont distribués par exemple). Une autre différence entre ces deux concepts est qu'un *conteneur* n'est pas une partie d'un composant, comme c'est le cas pour un *contrôleur*. Le concept de *contrôleur* est plus large puisqu'il regroupe également la notion de *Maison de composant* (gestion du cycle de vie) proposée par les modèles EJB et CCM. La description de l'utilisation et de la configuration hors des composants, de certains services non-fonctionnels, semble particulièrement intéressante, dans notre problématique, pour fixer certaines propriétés propres au déploiement (en particulier celles liées à l'ordonnancement), qui ne peuvent être fixées au moment de la programmation de composants (car l'ordonnancement nécessite une vision "globale" de l'architecture).

4.3.7 Intergiciel et Atelier de Génie Logiciel

Le facteur déterminant de l'utilisation d'une approche basée sur les composants est l'environnement logiciel supportant l'approche. Cet environnement logiciel peut être caractérisé en deux parties : l'environnement de développement (Atelier de Génie Logiciel, AGL) et l'environnement d'exécution (Intergiciel). Un ADL est le plus souvent couplée à une suite d'outils assimilable à un Atelier de Genie Logiciel (AGL). Cet AGL sert en premier lieu à la description et à la visualisation d'une architecture logicielle, mais il sert également à gérer d'autre préoccupations des ADLs, comme :

- La validation d'une architecture qui consiste à vérifier si une interconnexion de composants est valide.
- L'analyse (formelle ou par simulation) du comportement d'une application.
- La génération automatique du code "squelette" de l'architecture application.
- La programmation ou l'intégration d'implantations des composants de l'architecture.
- Le placement des composants, qui consiste à décrire la localisation des composants d'une application sur une infrastructure de déploiement (matérielle et système) comme c'est le cas dans MetaH. Ceci permet d'automatiser, partiellement, le déploiement d'une application.

Dans les modèles de composants ce sont les Intergiciels qui sont essentiels, même si certains modèles de composants, comme Fractal ou CCM, proposent des AGL pour faciliter leur utilisation. Un intergiciel a pour tâche minimale de permettre l'interopérabilité des composants logiciels, ce qui suppose au minimum de pouvoir les composer et de les rendre capables de s'exécuter et de communiquer (ces fonctionnalités peuvent être assumées au moins partiellement par un OS ou par une machine virtuelle). Les autres préoccupations d'un intergiciel sont :

- Automatiser, au moins partiellement, le processus de déploiement des composants sur un site, par exemple en fonction des informations issues d'un AGL, comme c'est le cas pour OpenCCM.
- Gérer un ensemble de services non-fonctionnels (ordonnancement, communications, persistance), afin de les adapter au contexte d'exécution et à la demande des composants "métier".

Une approche basée sur les composants, visant à soutenir un processus de développement complet doit être outillée d'un intergiciel et d'un AGL qui implantent, dans la mesure du possible l'ensemble des fonctionnalités listées. Optimalement, Intergiciel et AGL sont en interaction afin de faciliter les phases de déploiement, de simulation et de test.

Résumé :

Une approche basée sur les composants :

- permet de décrire des architectures logicielles par assemblage de composants définis indépendamment.

- est basée sur différentes entités :
  - des composants qui :
    - sont constitués de ports, requis ou fournis, typés par des interfaces
    - sont assemblés par l'établissement de connexion entre leurs ports.
    - peuvent être des composites (ils agrègent des composants).
    - définissent des dépendances non-fonctionnelles avec l'infrastructure de déploiement..
    - ont un comportement interne décrit via une notation formelle et/ou du code.
  - des connecteurs qui :
    - réifient des protocoles d'interaction.
    - sont utilisés pour assembler des composants en fonction de leurs rôles.
  - des configurations qui :
    - permettent d'abstraire une (sous-)architecture logicielle.
    - permettent de gérer le déploiement des composants agrégés (répartition, ordonnancement, etc.).
  - des conteneurs qui :
    - fournissent un environnement d'exécution à des composants.
    - permettent l'utilisation simplifiée de services non-fonctionnels.

- nécessite :
  - un environnement de déploiement, d'exécution et de communication (intergiciel).
  - un environnement d'édition des composants (Atelier de Génie Logiciel) : (modélisation, programmation, analyse formelle, description du déploiement, simulation, etc.) .

Chapitre 5

Positionnement

Suite aux diverses réflexions menées sur les propositions existantes, présentées dans les chapitres précédents, nous faisons ici le "point". Ce chapitre charnière nous permet de dégager les grandes lignes de notre proposition avant de rentrer dans le détail de celle-ci.

5.1 Les idées clés

Dans cette section nous présentons de façon globale les différentes idées qui ont inspiré notre proposition.

5.1.1 Processus de développement : du modèle à l'exécution

L'étude des démarches de développement a permis de faire la synthèse sur la façon d'organiser un projet de développement logiciel. Les concepts d'*itération* et de *phase* permettent d'organiser simplement le processus de développement. En effet, un processus de type "itératif" a pour l'avantage de répartir clairement les tâches des développeurs en fonction de leur expertise (expertise métier, architecte, programmeur, déployeur, testeur). Il fournit donc un cadre général qui clarifie la mise en commun des différentes expertises au sein d'un projet de développement logiciel.

Une organisation "itérative" a également pour avantage de formaliser les différentes étapes de raffinement de la description d'une application. Le raffinement doit permettre de mener les développeurs d'un modèle abstrait de l'application (i.e. le modèle reflétant l'expertise métier mise en jeu par le projet) jusqu'à un modèle suffisamment détaillé pour être exécutable (i.e. le code de l'application). Dans une telle approche, le modèle est donc la base à partir de laquelle est mené le projet de développement. Un principe que nous voulons promouvoir, est d'organiser le raffinement dans un cadre de développement cohérent, de telle manière que les efforts de réécriture des modèles soient minimisés.

Un processus de ce type doit également fournir un moyen d'assurer la traçabilité de l'évolution de l'application, c'est-à-dire la mise en correspondance des descriptions de l'application entre les différentes phases. En effet, il est souhaitable que les entités logicielles puissent être vues avec différents niveaux de détail suivant la phase du processus de développement dans laquelle elles sont manipulées. La traçablité impose aussi d'assurer la cohérence de la description d'une application entre les phases car, par exemple, des choix faits à la phase de programmation peuvent modifier (dans des limites à déterminer) le modèle issu de la phase de conception.

Afin de mettre en place de façon efficace les principes de raffinement et de traçabilité, la méthodologie doit être outillée en conséquence. Un environnement de développement doit permettre de visualiser le modèle de l'application à différents niveaux de détail. Il doit faciliter, contrôler et répercuter sur la description de l'application (si besoin est) les modifications faites à n'importe quel niveau de détail. Au final, il doit générer du code logiciel qu'un environnement d'exécution est capable

de déployer puis d'exécuter. L'existence de tels environnements permettrait de retirer un des bénéfices les plus importants des processus de développement "agiles", à savoir obtenir le plus rapidement possible du code logiciel à des fins de test.

5.1.2 Description : approche basée sur les composants

Partant de cette base méthodologique simple, une des questions essentielles était de savoir quelles notations utiliser durant le processus, pour modéliser et programmer un contrôleur de robot. Pour cela, la première étape était de comprendre les pratiques actuelles de développement des contrôleurs de robots afin d'en conceptualiser les besoins et limitations actuelles.

Pour répondre aux besoins et limitations, nous avons étudié un ensemble de propositions gravitant autour du monde des composants logiciels. Le choix, a priori, du paradigme des composants logiciel est influencé par les qualités qu'il véhicule en terme de réutilisation, de lisibilité et de modularité du logiciel. Le modèle d'assemblage des composants logiciels, par connexion de leur ports, est facile à appréhender car intuitif à utiliser (à l'image des modèles d'assemblage des composants électroniques ou mécaniques). Le concept d'interface est essentiel, car il permet de représenter et de catégoriser l'offre et l'utilisation de services indépendamment des composants qui implantent ou utilisent ces services. Le concept d'interface constitue la base d'une possible standardisation des échanges des composants (au sein d'un projet, d'une entreprise, d'un organisme de standardisation, etc.), essentielle, à terme, pour pouvoir réaliser l'intégration de composants issus de fournisseurs divers. La création d'une application revient donc à identifier différents services, à caractériser les échanges qu'ils induisent au travers d'interfaces et à sélectionner les composants dont les ports fournissent ou requièrent ces interfaces. Cette approche semble en conformité avec les besoins futurs de l'industrie et de la recherche en robotique.

Les approches basées sur les composants logiciels proposent, de plus, des solutions adaptées à la gestion des besoins inhérents à la conception d'architectures de contrôle :

- Elles apportent une certaine qualité dans la description des composants et des architectures de contrôle, via l'utilisation de notations formelles ou empiriques au dessus desquelles peuvent être réalisées des analyses/simulations. Le modèle de composition doit être sous-tendu, autant que possible, par des formalismes permettant aux développeurs de vérifier la validité d'une composition, à l'image de ceux proposés dans les ADLs.
- Elles apportent une partie des réponses à la description d'architectures logicielles suivant une approche systémique, à travers les notions de composant composite et de configuration.
- Elles apportent des solutions pour le déploiement des architectures de contrôle. La phase de déploiement est essentielle puisque ces architectures sont intrinsèquement dépendantes du matériel embarqué (capteurs, actionneurs et matériel de communication). D'autres part, la prise en compte de propriétés système de bas niveau est essentielle pour gérer certains des aspects "temps-réel" de ces architectures, en particulier l'ordonnancement des processus système.
- Elles apportent une solution pour gérer la complexité et la variété des interactions possibles entre entités de contrôle, à travers le concept de connecteur. Grâce aux connecteurs, il est possible de réutiliser indépendamment les entités logicielles qui encapsulent du code "métier"(les composants) et celles qui encapsulent du code lié à la gestion des interactions des composants (les connecteurs). Cela apporte globalement une plus grande souplesse dans le choix des protocoles utilisables pour connecter des composants issus de différents modèles d'architecture de contrôle (chaque modèle prédéfinissant et limitant en général les interactions possibles, à un ensemble prédéfini de protocoles spécifiques).

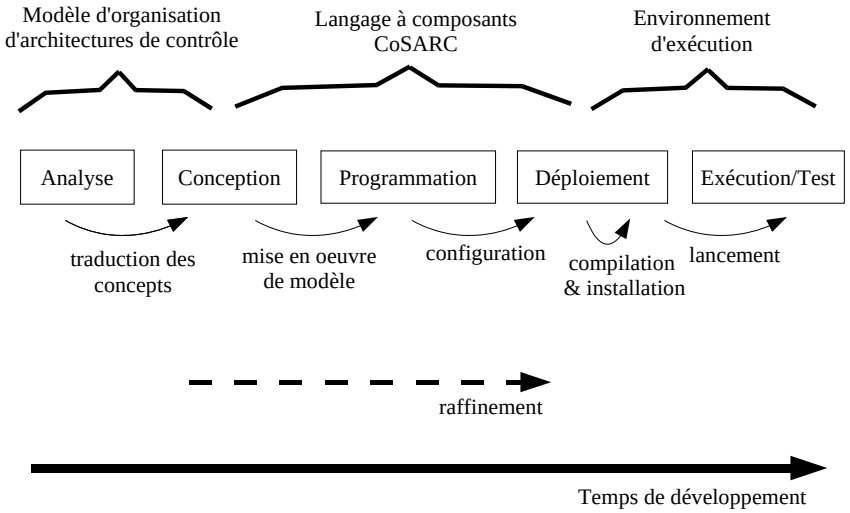


FIG. 5.1 – Processus de développement d'un contrôleur dans COSARC

5.2 Contenu de la proposition

5.2.1 Processus de développement

La méthodologie présentée dans ce mémoire de thèse se nomme CoSARC, acronyme anglais pour *COmponent-based Software Architecture of Robot Controllers*. Elle définit un processus de développement qui organise un projet de développement logiciel en plusieurs itérations successives qui permettent de gérer un projet de développement de façon incrémentale. Une même itération vise à mobiliser les développeurs afin qu'ils produisent ou corrigent une partie identifiée de l'application. Chaque itération est définie suivant un ensemble d'étapes distinctes et successives (cf. figure 5.1) appelées phases : analyse, conception, programmation, déploiement et test.

Ce processus de développement est très classique et selon nous relativement simple à appréhender et à appliquer. Notre intérêt scientifique consiste à définir précisément ce que les développeurs doivent faire à chacune des phases du processus, quelles notations ou outils ils doivent utiliser.

5.2.2 Modèle d'organisation des architectures de contrôle

Durant la phase d'analyse, les développeurs doivent identifier et organiser les concepts "métier" qu'ils auront à manipuler durant tout le processus de développement. Dans le cadre très particulier de la robotique, l'identification et l'organisation des concepts est un travail en soit très compliqué, puisqu'il oblige à prendre en considération l'intégration de connaissances et de savoir-faire de personnes ayant des compétences très diverses. Pour faciliter cette intégration nous proposons de conduire cette phase à partir de concepts décrits dans un modèle. A partir de ce modèle, nous proposons une démarche d'analyse la plus générique possible. Cette démarche permet aux développeurs d'identifier, à partir de leur besoins fonctionnels (i.e. ce que doit faire le robot) et contraintes matérielles (i.e. issues de la partie opérative du robot), les concepts importants qui seront traduits sous forme d'entités logicielles.

L'autre utilité de ce modèle est qu'il définit la façon d'organiser les entités logicielles d'une architecture de contrôle en fonction des concepts "métier" définis préalablement. Il est donc un support

conceptuel pour la phase de conception. Ce modèle reprend l'ensemble de nos réflexions sur l'organisation des architectures de contrôle, présentées à la fin du chapitre 3. Il est donc basé sur le découpage d'une architecture en différentes couches conceptuelles, en charge d'un "niveau de prise de décision". Chaque couche est constituée d'un ensemble d'entités de contrôle qui échangent des connaissances en interagissant. Evidemment, ce modèle d'organisation va plus loin en définissant les différents types d'entités de contrôle et les différents types d'interactions. Il définit également les types d'entités qui représentent les connaissances que le contrôleur possède sur le "monde" du robot et leurs relations avec les types d'entités de contrôle.

Ce modèle d'organisation des architectures de contrôle a été défini pour fournir des abstractions intuitivement compréhensibles et suffisamment génériques pour être adaptables à des robots de natures très différentes. D'autre part, il est suffisamment "neutre" pour que la démarche d'analyse ne soit pas dépendante d'outils, langages ou formalismes de modélisation précis, que nous considérons comme secondaires lors de cette phase.

5.2.3 Langage à composants

Les phases de conception et de programmation nécessitent l'utilisation de langages de modélisation et de programmation. La phase de conception consiste à décrire un modèle détaillé (structurel et comportemental) des constituants de l'architecture logicielle d'un contrôleur et la phase de programmation consiste à écrire le code d'implantation de ces constituants. Pour couvrir l'ensemble de ces phases, nous proposons un *langage à composants*. Les notations offertes par ce langage couvrent les phases de conception (modélisation), de programmation et de déploiement. Dans ce cadre, il intègre un mécanisme de *raffinement* simple qui permet de passer rapidement d'une phase à une autre. Tout au long du processus de développement, le langage CoSARC est utilisé pour écrire le modèle d'une architecture puis raffiner ce modèle jusqu'à obtenir une description complète et détaillée de chaque composant, à partir de laquelle du code exécutable peut être généré. Ainsi, selon la phase dans laquelle le langage CoSARC est utilisé, des aspects particuliers de l'architecture seront décrits. Pendant la phase de conception, le langage CoSARC sert à décrire l'ébauche de l'architecture du contrôleur : les composants sont vus comme des boîtes noires dont les ports sont connectés par des connexions. Pendant la phase de développement, le langage CoSARC permet d'écrire le code d'implantation des composants. Enfin, le langage CoSARC permet également de décrire la façon dont sera déployée l'architecture. Ainsi nous tentons de proposer un mécanisme de raffinement simple qui permette d'aller progressivement *du modèle à l'exécution*.

Nous qualifions le langage CoSARC de *langage à composants* car il tente de tirer le meilleur parti à la fois des ADLs (modélisation, analyse, génération de code à partir de modèles) et des modèles de composants (programmation, composition, interopérabilité) en plaçant les notions de composants et d'architecture au centre du processus de développement. Durant chacune de ces phases, les architectures et composants sont visibles et manipulables. Les caractéristiques essentielles de tout composant sont présentées de façon simplifiée dans la figure 5.2.

- *un composant* est une entité informatique exécutable qui exprime les services qu'il offre à travers un ensemble de *ports fournis* et exprime les services qu'il utilise à travers un ensemble de *ports requis*. Un composant contient un ensemble de *propriétés* internes, par exemple un ensemble d'*attributs* et de *méthodes*.

- *un port* est un point de connexion d'un composant. Chaque *port* est typé par une *interface*. Un *port fourni* permet à d'autres composants d'utiliser un service offert par un composant. Un *port requis* permet à un composant d'utiliser un service offert par d'autres composants. Un *port* est rattaché à un ensemble de *propriétés* internes du composant.

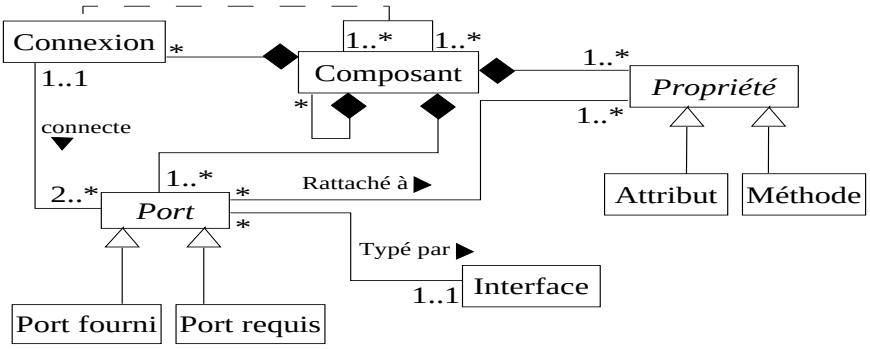


FIG. 5.2 – Modèle de composant simplifié de CoSARC

- *une interface* est un contrat qui contient la définition de l'offre ou de l'utilisation d'un service. L'offre (respectivement l'utilisation) d'un service peut être composé d'un ensemble d'offres (respectivement d'utilisations) de sous-services : une interface peut être composée d'un ensemble de sous-interfaces. Une même *interface* peut être référencée par un nombre quelconque de ports *fournis* ou *requis* (potentiellement aucun), appartenant à un nombre quelconque de composants.
- *une connexion* entre les ports de composants permet d'exprimer leur assemblage (on dit alors que les ports sont connectés). Lorsqu'il sont assemblés, les composants interagissent autour de l'offre et de l'utilisation de services (i.e. en fonction des interactions définies au niveau des interfaces). Une connexion est établie si les interfaces des ports connectés sont conformes. Nous verrons qu'il existe différentes sortes d'interfaces et donc différentes relations de conformité.
- Un *composant* peut être un *composite*, c'est-à-dire qu'il agrège un ensemble de *composants* assemblés ou non. Les *composants* et les *connexions* agrégés au sein d'un composite, sont référencés par un ensemble d'*attributs* du composite.

L'utilisation du langage CoSARC se fait via deux environnements. L'environnement de programmation est un atelier de génie logiciel (AGL) qui permet de décrire les composants, l'architecture, le déploiement et de générer des fichiers correspondants. L'environnement d'exécution est chargé d'exécuter le code contenu dans ces fichiers, c'est-à-dire déployer une architecture logicielle (compilation et installation des composants) et d'autre part exécuter les composants et permettre leur communication.

5.3 Orientation de la proposition

Au travers des choix effectués, nous faisons la distinction entre préoccupations indépendantes de la plateforme (modèle d'organisation des architectures de contrôle) et préoccupations spécifiques à la plateforme (langage à composants). Le modèle d'organisation d'architectures de contrôle permet à un développeur de raisonner sur une architecture de contrôle en terme de concepts métiers. Il peut être vu comme un meta-modèle indépendant des plates-formes (meta-PIM dans le jargon MDA) permettant de décrire un modèle ébauche d'une architecture. Le langage à composants permet à un développeur de décrire une architecture à composants à partir de notations qui sont compilées et exécutées par des outils logiciels spécifiques. Les notations proposées par le langage sont décrites au sein d'un meta-modèle, que l'on peut considérer comme un meta-modèle spécifique à une plate-forme (meta-PSM dans le jargon MDA) puisque ces notations ont pour vocation d'être transformée en code exécutable. Si nous ne nous intéresserons pas aux problèmes de transformation automatique d'un

modèle PIM en modèle PSM, problème encore ouvert à ce jour, nos travaux doivent être pensés pour prendre en compte cette possibilité future.

Nous pensons que cette dissociation va augmenter la portée de chacune des deux propositions. D'une part le modèle d'organisation d'architecture de contrôleur n'est pas limité par l'usage de technologies spécifiques, il doit pouvoir être utilisé avec différents environnements de développement logiciel. D'autre part, le langage à composant n'est pas borné à un modèle d'organisation d'architecture de contrôle, il doit pouvoir être utilisé dans différents contextes applicatifs. Cette indépendance se traduit et se justifie suivant deux points de vue :

- Les modèles d'organisation doivent être indépendants des technologies utilisées pour mettre en œuvre les architectures de contrôle. Exprimer ces modèles d'organisation à partir d'un langage de modélisation standard comme UML, permet de mieux comprendre et diffuser les principes guidant l'organisation d'une architecture de contrôle. Cela permet également de plus facilement comparer différents modèles d'organisation et les faire évoluer en fonction de l'expérience des développeurs (puisqu'une modification d'un modèle d'organisation n'a pas de répercution technologique directe).
- Les plates-formes technologiques doivent être indépendantes des modèles d'organisation. Cela permet, par exemple, de comparer la performance de différentes plates-formes technologiques concurrentes (en terme de fonctionnalités offertes, de facilité d'utilisation, de réutilisation, etc.) pour mettre en œuvre les principes d'organisation d'un même modèle d'architecture. Cela permet également de ne pas avoir à modifier une plate-forme dès lors qu'un modèle d'organisation évolue, seules les "règles" de traduction (formalisées ou non) devant être changées.

Pour les phases de conception et de programmation, nous avons fait le choix de définir un langage à composants dédié (par opposition à l'utilisation d'UML et de langages de programmation existants), car nous voulons adapter le développement aux besoins et aux compétences des praticiens du domaine d'application visé. Pour décrire le meta-modèle du langage à composant CoSARC, deux options s'offraient alors à nous. La première consistait à étendre le meta-modèle d'UML (i.e. étendre le langage UML) et la seconde à créer un meta-modèle propre à partir du MOF (i.e. définir un langage dédié). Proposer un langage dédié peut paraître étrange à cette époque où l'homogénéisation est courante, souvent induite par des langages de programmation populaires commes Java ou des langages de modélisation standardisés comme UML. Pourtant, l'avantage de cette approche est qu'elle permet d'avoir un meta-modèle à la fois détaillé et compréhensible pour le plus grand nombre, ce qui n'aurait certainement pas été le cas en proposant une extension d'UML. Ainsi notre meta-modèle PSM est débarassé de concepts non-essentiels présents dans le meta-modèle d'UML. Cette approche facilite également la transition entre les phases de conception et de programmation puisque le même langage est utilisé pendant ces phases. Cela permet d'éviter une étape de génération de squelettes de code à partir de modèles UML et permet d'éviter que les développeurs ne manipulent plusieurs notations différentes (langage de modélisation et langage de programmation).

Dans la seconde partie du mémoire, nous présentons en détail les deux facettes de notre proposition. Le chapitre 6 présente le modèle d'organisation des architectures de contrôle. Le langage à composants CoSARC est présenté dans le chapitre 7 et son environnement d'exécution est présenté dans le chapitre 8. Enfin, le chapitre 9 propose un exemple d'application de la méthodologie au développement d'un robot manipulateur mobile.

Deuxième partie

La méthodologie CoSARC

Chapitre 6

Modèle d'organisation d'architectures de contrôleurs

Ce chapitre présente un modèle d'organisation pour les architectures de contrôle, utilisé comme support de l'analyse d'un contrôleur de robot mobile. Les différents concepts à la base de l'organisation d'une architecture logicielle d'un contrôleur, sont expliqués à travers des modèles UML et des définitions dans les sections suivantes. La première section présente l'organisation "en couches" des entités constituant une architecture. La deuxième section présente la démarche d'analyse reposant sur le modèle d'organisation. Enfin, les troisièmes et quatrièmes sections présentent respectivement les diagrammes de classes représentant la vue statique et les diagrammes de séquences représentant la vue dynamique de ce modèle.

6.1 Organisation hiérarchique et systémique

Le modèle d'organisation d'architectures de contrôle (que nous nommerons plus simplement modèle d'architecture) est basé sur une décomposition d'une architecture en un ensemble de "couches". Chaque couche représente un niveau de prise de décision. Les couches sont organisées entre elles suivant les critères (abstraction des données, complexification de la prise de décision, contrainte temporelles de réaction) énoncés dans la section 3.8.3. Chaque entité du modèle (cf. fig. 6.2) représente une responsabilité que doivent assumer une ou plusieurs entités de contrôle (ou sous-système) d'une architecture. Ils permettent d'exprimer des concepts "métier" aidant les développeurs à décrire l'organisation du contrôle du robot. Les relations entre les entités du modèle expriment des relations entre les entités de contrôle qui assument les responsabilités correspondantes. Ces relations expriment des envois d'*intentions* (e.g. objectifs à atteindre, vecteurs de commande) et des réceptions d'*observations* (permettant de déduire les obligations liées au contexte). Le modèle d'organisation d'architecture reprend les principes d'organisation présentés dans la section 3.8.4. Il peut être vu comme un patron d'analyse/conception, c'est-à-dire comme un guide pour diriger l'organisation au sein d'une architecture de contrôle.

Le modèle d'architecture proposé (fig. 6.2) est inspiré des concepts déjà présents, entre autres, dans le modèle d'architecture proposé par le LIRMM. Il vise en particulier, à mettre en adéquation les concepts qui permettent de décrire l'organisation du contrôle avec et les entités logicielles d'une architceture. Pour illustrer nos propos, nous prenons pour exemple le cas d'un contrôleur de robot manipulateur mobile terrestre. Dans les entités présentés par la suite, nous nommons *ordre* une *intention* émise par les entités de contrôle des couches hautes d'une architecture. En fonction de la couche, nous donnons différents noms aux ordres : *ordre de mission* dans la couche la plus haute, *ordre de ressource* et *ordre primitif* dans les couches intermédiaires. Dans les couches les plus basses

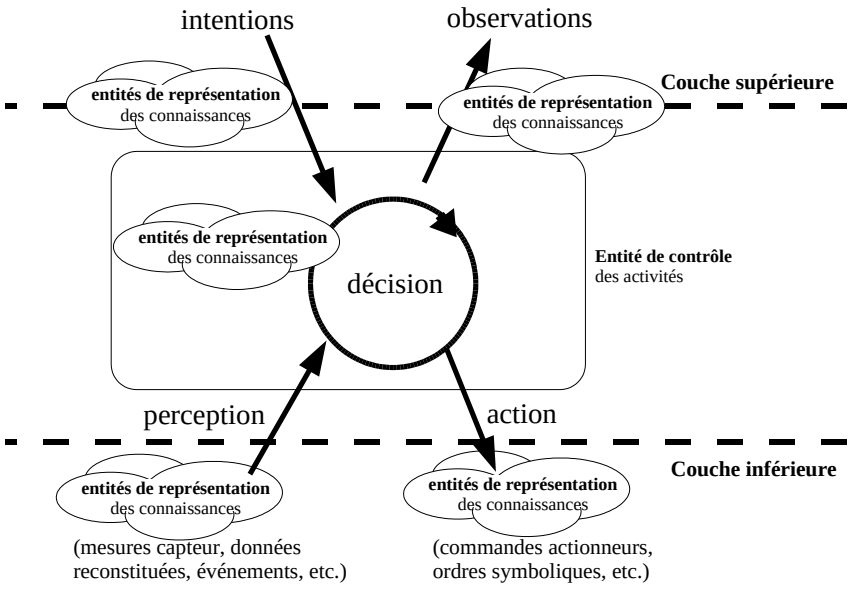


FIG. 6.1 – Modèle générique du comportement et des intéractions d'une entité de contrôle

nous parlerons de *consignes* (au niveau des asservissements) et de *vecteur de données de commande* (au niveau des entrées/sorties) pour désigner les *intentions* émises.

L'entité la plus globale dans une architecture de contrôle est un *Contrôleur de robot*. Elle correspond simplement à l'intégralité du contrôleur d'un robot. Elle agrège un *Superviseur Global*, des *Ressources robotiques*, des *Générateurs d'événements* et des *Contrôleurs d'entrées/sorties*. La nécessité d'abstraire l'architecture de contrôle d'un robot par une entité s'impose dès lors que les missions robotiques comportent de multiples intervenants : on peut alors représenter simplement les robots et les *Plates-formes opérateurs* participant à la mission.

Un *Contrôleur d'Entrées/Sorties* est une entité qui procède périodiquement à l'échantillonnage des mesures d'un ensemble de capteurs et/ou à l'application des données de commande à un ensemble d'actionneurs. Un *Contrôleur d'Entrées/Sorties* permet à d'autres entités de contrôle d'agir et d'obtenir des informations sur la partie opérative et sur l' environnement. Il peut également encapsuler un mécanisme d'arbitrage (e.g. de sommation de vecteurs de données de commande) dans le cas où les développeurs adoptent une approche comportementale dans la couche supérieure (arbitrage de *commandes* concurrentes). Ils sont situés dans la couche la plus basse d'une architecture, qui se situe conceptuellement au-dessus de l'instrumentation embarquée sur un robot, et dans laquelle l'activité des entités est (conceptuellement si ce n'est techniquement) synchrone (piloté par une horloge).

Une *Ressource robotique* (que nous nommerons par la suite simplement *Ressource*) est une entité qui agrège un ensemble d'entités de contrôle coordonnées, relatives au contrôle d'un ensemble commun d'éléments matériels de la partie opérative. Une *Ressource* contrôle des éléments matériels embarqués de façon indépendante et exclusive du reste du matériel embarqué. Une *Ressource* est donc vue comme un contrôleur d'un sous-ensemble de la partie opérative d'un robot. Des exemples de *Ressources*, au sein d'un `contrôleur de robot manipulateur mobile` sont le `Manipulateur` (contrôle du bras mécanique) et le `Mobile` (contrôle du véhicule). Le rôle d'une *Ressource* est d'assurer la réalisation des *ordres* (correspondant aux *intentions*) provenant d'un *Superviseur global*, ou de l'informer de son état d'avancement dans sa mission. Une *Ressource* est décrite à partir d'un ensemble d'entités :

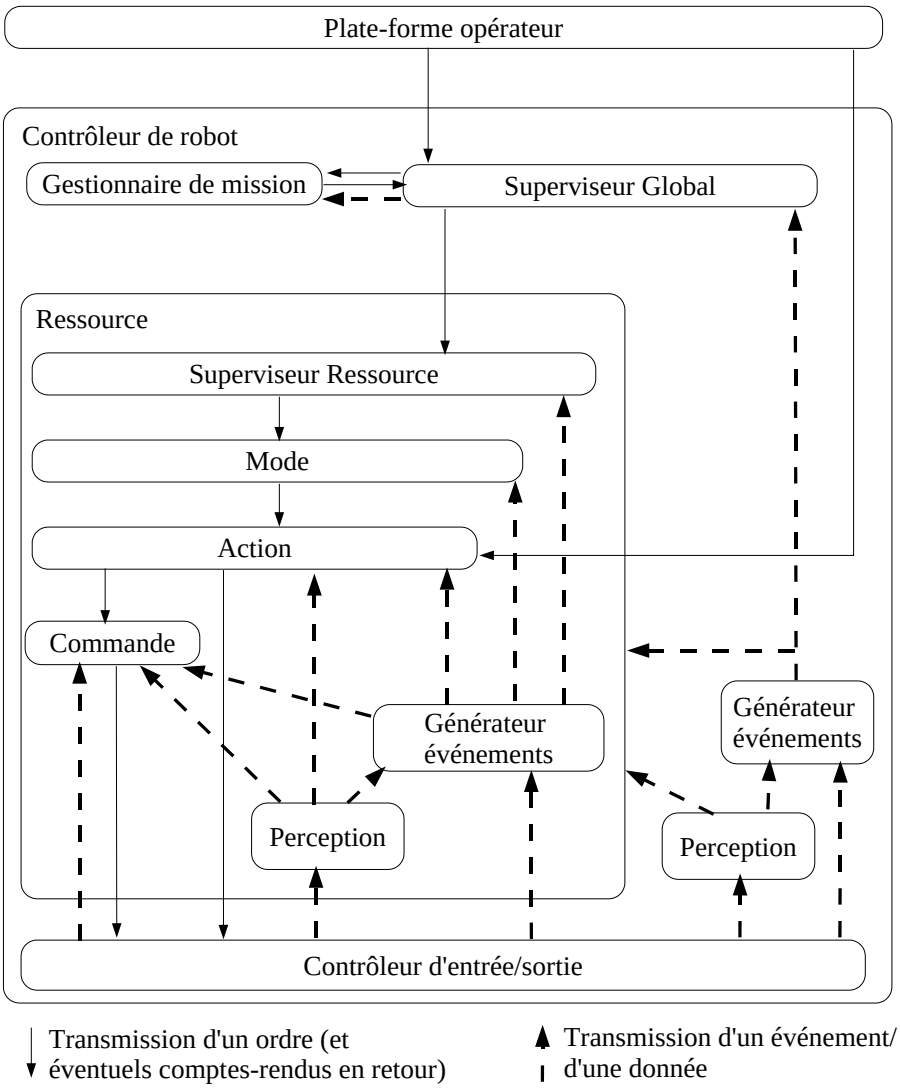


FIG. 6.2 – Modèle informel d'organisation d'une architecture de contrôle : vision hiérarchique et systémique

- Un *Générateur d'événements* est une entité chargée de détecter l'occurrence d'un ensemble d'*événements* (phénomènes d'origine proprioceptive ou exteroceptive) et d'en notifier les entités en ayant fait la demande. Dans l'exemple du robot manipulateur mobile, des *Générateurs d'événements* servent à détecter des obstacles dans la trajectoire du véhicule, des configurations singulières du bras mécanique, etc.
- Une *Perception* est une entité chargée de transformer périodiquement des données brutes (issues de l'échantillonnage des capteurs) en des données de plus haut niveau d'abstraction et de les proposer à d'autres entités. Elle sert, par exemple, au traitement des données issues de caméras embarquées sur le robot, à de la fusion de données ou plus généralement à l'observation d'état (reconstruction de grandeurs non accessibles directement par la mesure).
- Une *Commande* est une entité chargée de lire des données (issues de *Contrôleurs d'Entrée/Sortie*, de *Perceptions* ou de *Générateurs d'événements*) et de générer en conséquence des données de commande au *Contrôleurs d'Entrées/Sorties* afin qu'ils les appliquent. Une *Commande* applique une loi de commande, une commande événementielle ou une commande hybride qui permet de piloter un sous-ensemble identifié de la partie opérative d'un robot. Le terme *Commande* généralise les termes d'*asservissement*, souvent utilisé dans les architectures hiérarchiques, et celui de *comportement réactif*, souvent utilisé dans les architectures comportementales.

Les *Générateurs d'événements*, les *Perceptions* et les *Commandes* se situent dans la couche directement supérieure à celle contenant les *Contrôleurs d'Entrées/Sortie*. On peut apparenter cette couche à la couche fonctionnelle présente dans de nombreux modèles. Ils sont à la frontière entre un monde synchrone et un monde asynchrone, une partie de leur comportement étant périodique (boucle de commande ou d'observation) et l'autre partie étant apériodique (observations produites et traitement des intentions reçues fait de façon apériodique). Ces entités interagissent avec un monde où les données sont essentiellement numériques et avec un monde où les données sont plus complexes et symboliques.

- Une entité *Action* est en charge de réaliser un *ordre primitif*, en s'appuyant sur un ensemble de *Commandes*. Le positionnement du préhenseur du bras mécanique est un exemple d'*Action* utilisée par la *Ressource* `Manipulateur`. Le déplacement du véhicule vers une position dans l'environnement est un exemple d'*Action* utilisée par la *Ressource* `Mobile`. Une *Action* gère l'adaptation dynamique de la ou des *Commandes* actives, afin de donner à la *Ressource* le comportement adéquat (pour réaliser l'*ordre primitif*). Pour réaliser cette adaptation, une *Action* s'appuie sur les services offerts par les *Commandes* qu'elle utilise. Ces services permettent de configurer dynamiquement les paramètres des *Commandes* qui sont, par exemple, la consigne à atteindre, le jeu de gain, la valeur de pondération du vecteur de données de commande généré ou du vecteur de données de perception, la valeur initiale des termes d'une loi, etc. Une *Action* s'appuie également sur des *données d'état* générées par des *Perceptions* et/ou des *événements* provenant de *Générateurs d'événements* et/ou des *comptes-rendus* d'exécution ou d'erreur de *Commandes*, afin de déduire son contexte d'exécution courant. A partir de ce contexte d'exécution, une *Action* décide des commutations et du paramétrage des *Commandes* actives. Par exemple, une *Action* chargée de déplacer le bras mécanique du robot, doit mettre en place un mécanisme de commutation si certaines des *Commandes* utilisées sont sensibles aux singularités. Dans ce cas, elle interagit avec un *Générateur d'événements* qui l'alerte de l'approche ou de la sortie d'une configuration singulière. Une fois terminée, une *Action* doit remettre les éléments matériels sous son contrôle dans une position stable (à partir de laquelle d'autres *Actions* pourront être réalisées). Si ces éléments matériels requièrent d'être asservis en permanence (e.g. le bras mécanique), alors elle doit activer une *Commande* spécifique, choisie pour leur stabilisation (e.g. compensation de la gravité du bras mécanique). Enfin une *Action* doit toujours alerter la ou les entités des couches supérieures qui la contrôle, de l'impossibilité de réaliser l'*ordre primitif*.

- Un *Mode* représente un fonctionnement particulier d'une *Ressource* selon la nature de la relation de cette *Ressource* avec d'autres intervenants (i.e. d'autres *Ressources*, d'autres robots, des opérateurs humains). Par exemple un `Mode autonome` représente un fonctionnement isolé d'une *Ressource* (qui est capable de réaliser des objectifs de haut niveau d'abstraction) ; un `Mode téléopéré` représente un fonctionnement d'une *Ressource* directement pilotée par un opérateur (qui émet des consignes d'asservissement) ; un `Mode Coopératif` représente un fonctionnement d'une *Ressource* en relation avec une ou plusieurs autres *Ressources* (e.g. le fonctionnement du `Mobile` au sein d'une flotille, dans une relation meneur/suiveur [54]). Une même *Ressource* peut fonctionner dans plusieurs *Modes* différents, mais de manière exclusive (un seul *Mode* actif à un moment donné). Chaque *Mode* est capable de réaliser un ensemble prédéfini d'*ordres de ressource*, en utilisant un ensemble prédéfini d'*Actions*. Il traduit chaque *ordre* qu'il reçoit en une séquence d'*ordres primitifs* que ces *Actions* doivent successivement réaliser. L'avantage de distinguer des *Modes* différents pour des *Ressources* est la possibilité d'exprimer l'activation simultanée de *Ressources* dans *Modes* différents. Par exemple, le `Manipulateur` est en `mode autonome` pour effectuer un transport de charge amorti et le `Mobile` est en `mode téléopéré` pour que l'opérateur l'amène à un point donné.
- Un *Superviseur de Ressource* est chargé de la supervision et du contrôle des *Modes* d'une *Ressource*. Il met en œuvre les mécanismes d'activation et de désactivation des *Modes* de la *Ressource*, en fonction des *ordres de ressource* transmis par le *Superviseur Global* et/ou des *événements* qui lui sont notifiés par des *Générateurs d'événements*. Il garantit la cohérence d'activation des *Modes* (deux *Modes* d'une même *Ressource* ne peuvent être actifs en même temps). Les *Superviseur de Ressource*, *Mode* et *Action* peuvent être conceptuellement regroupés dans une couche exécutive, par analogie avec plusieurs modèles d'architectures existants.

Un *Superviseur Global* est chargé de contrôler et synchroniser l'activité d'un ensemble de *Ressources*. Il n'existe qu'un unique *Superviseur Global* pour un *Contrôleur de robot*. Il est capable de réaliser un ensemble d'*ordres de mission*, qu'il traduit sous la forme d'*ordres de ressource* qu'il envoie aux *Ressources* (et donc aux *Superviseurs de Ressource*) concernées. Il supervise la réalisation des *ordres de mission* reçus et gère les contraintes considérées comme globales pour le robot (e.g. son énergie disponible), via l'utilisation d'un ensemble de *Générateurs d'événements*. Lorsqu'il reçoit un *ordre de mission* qu'il ne peut directement traduire sous la forme d'un *ordre de ressource*, il interagit avec un *Gestionnaire de Mission*.

Un *Gestionnaire de Mission* est chargé de la planification des missions d'un robot. Il traduit un *ordre de mission* provenant d'un *Superviseur Global* en un plan qui décrit les différents *ordres de ressource*s que chaque *Ressource* doit réaliser (séquentiellement pour chaque *Ressource* et parallèlement entre chaque *Ressource*). Le *Gestionnaire de Mission* envoie ce plan (ou des mises à jour de ce plan) au *Superviseur Global* qui se charge alors de le réaliser. Le *Superviseur Global* lui notifie des *événements* (liés au déroulement de la mission) qui sont susceptibles de remettre en cause la planification qu'il a initialement réalisée, et à partir desquels il pourra décider de replanifier la mission afin de mettre à jour le plan.

La *Plate-forme opérateur* est l'organe décisionnel de plus haut niveau dans une mission, elle constitue le point d'interface entre un opérateur humain et les robots participants à la mission. Elle est chargée de gérer et superviser les opérations réalisées (e.g. téléopération de bas niveau) ou demandées (e.g. réalisation d'un *ordre de mission*, suivi de l'avancement de mission). D'autre part, la *Plate-forme opérateur* peut également posséder des fonctionnalités de planification et de supervision dans le cas où la mission implique la coopération d'une flotte de robots. Elle forme, avec le *Superviseur Global* et le *Gestionnaire de mission*, la couche décisionnelle du robot, par analogie avec les modèles d'architecture existants.

Dans ce modèle d'architecture il existe plusieurs couches de planification, de supervision et de

contrôle des requêtes. Le mécanisme de décision d'une entité varie en complexité et en abstraction suivant la couche dans laquelle elle se trouve, de l'application d'une loi de commande dans les couches basses de l'architecture à l'application d'un mécanisme de planification et de supervision globale dans la couche la plus haute. Le concept de *Ressource* est essentiel pour définir et décrire des sous-parties d'un contrôleur de manière indépendante. Il permet aux développeurs de se concentrer sur le contrôle d'un sous-ensemble de la partie opérative d'un robot, ce qui permet de ne considérer qu'un sous-ensemble de ses degrès de libertés.

Ce modèle d'architecture permet de concevoir des architectures mixtes. Les deux critères pour considérer une architecture comme mixte sont présents. Premièrement, le transfert d'information entre les entités de contrôle peut "sauter" des couches : les *événements* générés par les *Générateurs d'événements* peuvent être transmis directement à n'importe quelle couche supérieure qui est alors chargée de prendre en compte ces événements pour décider de la réaction à effectuer. Le saut de couches lors du transfert d'information peut également se faire des entités des couches hautes vers les entités des couches basses, mais dans des cas spécifiques. C'est par exemple le cas lorsqu'une relation de télé-opération directe est établie : la *Plate-forme opérateur* envoie directement des consignes à une entité *Action* qui se charge de gérer les contraintes liées aux communications ou aux obstacles dans l'environnement. Deuxièmement, ce modèle autorise (mais n'impose pas) la conception d'une architecture suivant une approche à la fois comportementale (vision émergente de l'intelligence) et délibérative (vision "formelle" de l'intelligence). En effet, une *Action* peut décider d'activer simultanément plusieurs *Commandes* et de décider des valeurs de pondération associées au *vecteurs de données de commande* qu'elles génèrent (elle configure pour cela un ou plusieurs *Contrôleurs d'Entrées/Sorties*). Ces vecteurs sont alors sommés et pondérés au niveau des *contrôleurs d'Entrées/Sorties*. Au delà de la couche contenant les *Actions*, l'architecture est conçue selon une approche hiérarchisée. Dans ce cas, on suppose que pour certaines entités, comme les *Modes* et le *Gestionnaire de Mission*, des techniques d'intelligence artificielle peuvent être utilisées, par exemple pour la planification. Notons que dans la suite de ce mémoire, nous limiterons notre réflexion au "saut d'information" entre les couches sans aborder des problèmes d'arbitrage de *Commandes* concurrentes. Nous nous focalisons, au niveau des entités *Actions*, uniquement sur des problèmes de commutation et de reconfiguration d'une unique *Commande* active à un moment donnée.

Nous présentons dans la section suivante la démarche de conduite de la phase d'analyse, qui se base sur les entités présentées. Cette démarche a pour but d'amener les développeurs à décomposer l'architecture d'un contrôleur suivant ce modèle d'organisation, c'est-à-dire identifier la responsabilité que va assumer chaque entité de contrôle ou (sous-)système.

6.2 Conduite de la phase d'analyse

Identifier, dans un projet robotique, les différentes entités correspondant à celles décrites dans le modèle d'architecture n'est pas chose aisée, c'est pourquoi nous proposons une démarche pour conduire la phase d'analyse. Celle-ci se décompose en un ensemble d'étapes s'enchaînant de façon cyclique comme le montre la figure 6.3. Plusieurs cycles peuvent être nécessaires avant d'obtenir une analyse suffisamment précise de l'architecture d'un contrôleur.

6.2.1 Identification des services rendus

La première étape consiste à définir les services que peut rendre un robot. En premier lieu, il est nécessaire de définir l'utilité du robot et les différents *ordres de mission* qu'il est capable de réaliser, les *comptes-rendus* qu'il est capable de donner et les *événements* qu'il est capable de notifier. En

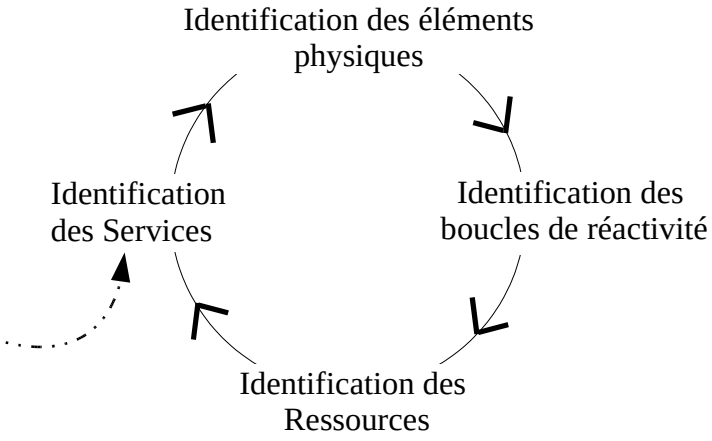


FIG. 6.3 – Cycle d'analyse d'un contrôleur

second lieu, il faut identifier les contraintes que doit respecter le robot pendant la réalisation de ses services.

Par exemple les services attendus du robot manipulateur mobile sont les suivants : la capacité de cartographier l'environnement ; la capacité d'identifier et de positionner dans l'environnement des objets de type connu (e.g. des objets cylindriques), objets ayant été désignés au robot par l'opérateur (le robot est donc chargé de l'extraction du contour et du positionnement), etc. Le robot manipulateur mobile peut émettre des *événements* signalant un changement d'état significatif de sa partie matérielle ou des *comptes-rendus* signalant l'impossibilité de réaliser un *ordre*. Le rôle de l'opérateur est donc de contrôler le robot à distance par des *ordres* représentant des demandes de réalisation d'objectifs de haut niveau d'abstraction (demande de positionnement d'objet) ou des *ordres* de demande de téléopération de bas niveau (téléopération directe du véhicule par exemple). Quel que soit l'*ordre* reçu, la contrainte que nous décidons d'imposer au robot est qu'il doit toujours éviter les obstacles et à aucun moment l'opérateur ne doit pouvoir l'obliger à réaliser des *ordres* contradictoires ou entraînant potentiellement sa destruction.

Une fois les services définis, les développeurs doivent identifier les caractéristiques physiques d'un robot.

6.2.2 Identification des caractéristiques physiques du robot

Lors de cette étape, les développeurs doivent déterminer quels éléments physiques constituent le robot. Il s'agit d'identifier, pour chaque service que le robot est susceptible de rendre, les éléments matériels de sa partie opérative, à partir desquels ce service peut être réalisé. Dans l'exemple du robot manipulateur mobile, sa partie opérative peut être décomposée en :

- un bras mécanique, nécessaire à la réalisation des tâches de manipulation d'objets dans l'environnement.
- un véhicule, nécessaire à la réalisation des déplacements du robot.

Les développeurs doivent alors définir la façon dont ces éléments matériels sont connectés les uns aux autres. Il faut, par exemple, définir la connexion de la base du bras mécanique avec un point de fixation sur le véhicule. Selon le degré de précision souhaité, un élément matériel peut être décomposé en un sous-ensemble d'éléments matériels connectés entre eux. Par exemple, le bras mécanique peut

être décomposé en un ensemble d'articulations, de segments, une base et un préhenseur. De la même façon le véhicule peut être décomposé en un chassis, des essieux, un moteur, des roues, etc.

Dans un même temps, il est nécessaire de définir l'ensemble des capteurs et actionneurs embarqués. A partir des capteurs, les développeurs savent quel type d'information ils pourront utiliser au sein d'un contrôleur pour évaluer l'état de la partie opérative du robot ou de son environnement. A partir des actionneurs, les développeurs savent quels moyens utiliser pour contrôler le robot. Par exemple, le robot manipulateur mobile a besoin de percevoir son environnement ; les développeurs pourront choisir d'utiliser comme capteur une caméra embarquée ou un système de faisceaux d'ultrasons. Ce choix des capteurs/actionneurs va influencer radicalement la façon dont le contrôleur va réaliser les services.

6.2.3 Identification des boucles de réactivité directes

L'identification des boucles de réactivité directes, appelées *Commandes* dans notre modèle d'architecture, est à la base de l'identification des *Ressources*. Il s'agit, en fonction des éléments matériels identifiés à l'étape précédente, de trouver des stratégies de commande adéquates au contrôle de la globalité ou de sous-ensembles de ces éléments.

Cette étape d'analyse est difficile, car elle demande une solide expertise en automatique et en robotique. En effet, elle demande une bonne connaissance des stratégies de commande existantes (lois de commande, commandes événementielles et commandes hybrides). Elle peut également demander des compétences dans la définition de nouvelles stratégies de commande. Pour le contrôle du bras mécanique du robot manipulateur mobile, nous avons retenu un ensemble de lois de commande [34] qui permettent de lui imposer différents comportements (déplacement vers un point dans son espace atteignable, recherche de contacts avec des parois, stabilisation avec compensation de la gravité, etc.). Nous y avons également ajouté une commande événementielle permettant de positionner ou d'enlever les freins du bras mécanique.

Une fois que les stratégies de commande utilisées ont été définies, on associe simplement une boucle de réactivité directe à chacune d'entre elles. Ainsi sont définies les entités *Commandes* disponibles.

6.2.4 Identification des Ressources

La décomposition d'un contrôleur en terme de *Ressources Robotiques* est essentiel à la bonne compréhension de la façon dont va être géré le contrôle du robot dans les couches les plus hautes. L'identification de l'ensemble des *Ressources* constituant un robot est une tâche complexe car bien souvent relative au point de vue qu'ont les développeurs sur la façon d'organiser le contrôle. Il est difficile de donner un guide de conduite généraliste et applicable pour tout projet, nous donnerons donc uniquement les "grandes lignes" quant à la façon de définir les *Ressources* d'un contrôleur. Lors de cette étape, le but principal est de faire coïncider les résultats issus des étapes d'analyse précédentes : les services (e.g. *ordres*, *événements*) que le robot doit être capable de rendre, les éléments matériels qui le constituent, les boucles de réactivité directes identifiées. Nous donnons par la suite un ensemble de "bonnes pratiques" à suivre lors de cette étape :

- Identifier les parties matérielles d'un robot qui peuvent être intuitivement considérées de façon indépendantes par un observateur humain. Les critères de cette première sélection sont en général morphologiques et/ou fonctionnels. Même s'il restent subjectifs, ces critères permettent, dans les premières étapes d'analyse, de faire une ébauche des *Ressources* contrôlant chaque sous-ensemble d'éléments matériels identifié. Par exemple, le robot manipulateur mobile ayant un véhicule et un bras mécanique, les critères morphologiques et fonctionnels nous permettent de dire qu'il existe une Ressource `Mobile` correspondant au contrôle indépendant du véhicule, et une *Ressource* `Manipulateur` correspondant au contrôle indépendant du bras mécanique.

Evidemment, ces critères ne permettent pas de résoudre tous les problèmes : si une caméra orientable est embarquée sur le véhicule, doit-on considérer qu'elle est contrôlée par la *Ressource* `Mobile` ou doit-on considérer qu'elle est controlée par une autre *Ressource*, par exemple une *Ressource* `Vision` ? Il s'agit d'un choix architectural à part entière, puisque les deux solutions sont possibles.

- Identifier l'ensemble des boucles de réactivité directes utilisées au sein d'une *Ressource*. Chaque *Ressource* étant relative au contrôle d'une partie identifiée de l'équipement du robot, il s'agit de rattacher les *Commandes* permettant de contrôler ces éléments aux *Ressources* déjà identifiées. Inversement, il peut s'agir aussi d'identifier de nouvelles *Ressources* en fonction des *Commandes* déjà disponibles pour le contrôle de certains éléments matériels. Dans l'exemple du robot manipulateur mobile, le bras mécanique sert à toutes les tâches de manipulation (placement dans l'espace, saisie, support de charge). DeS *Commandes* permettant la réalisation de ces tâches de manipulation ont été définies, elles sont donc naturellement rattachées à la *Ressource* `Manipulateur`. De la même façon, le véhicule motorisé va servir à réaliser les déplacements et une *Commande* permettant de contrôler les déplacements du véhicule a été définie. Cette *Commande* est donc naturellement rattachée à la *Ressource* `Mobile`.

- Identifier les *Modes* de chaque *Ressource*. L'identification d'un *Mode* pour une *Ressource* donnée est faite en fonction de la nature de son fonctionnement en relation avec d'autres intervenants. Si la *Ressource* fonctionne de manière isolée elle possède un *Mode autonome* qui définit l'ensemble des *ordres de ressource* qu'elle est capable de réaliser de façon autonome. Si elle doit être pilotée par un opérateur, elle possède un *Mode téléopéré* qui définit l'ensemble des *ordres de ressource* qu'elle est capable de réaliser en relation avec un opérateur humain, etc. Par exemple, le véhicule permet au robot de se déplacer et d'explorer son environnement, nous considérons donc que le `Mobile` possède un *Mode autonome* qui définit des ordres d'exploration et de déplacement. Le véhicule peut être téléopéré directement par un opérateur humain, nous considérons alors que le `Mobile` possède un *Mode téléopéré* qui définit des ordres liés à la téléopération. L'identification des différents *Modes* d'une *Ressource* consiste, à cette étape, à catégoriser les *ordres de ressource* que peut réaliser chaque *Mode* de cette *Ressource*. Il faut garder à l'esprit qu'une *Ressource* est dans un *Mode* à un moment donné et qu'elle ne peut réaliser que les *ordres de ressource* réalisables dans ce *Mode*.

- Identifier les différentes *Actions* utilisées par chaque *Mode* pour réaliser les différents *ordres de ressource* qu'il définit. Chaque *Action* est définie à partir des *Commandes* identifiées pour cette *Ressource*. La règle à suivre est que l'*ordre primitif* réalisé par une *Action*, doit être intuitivement compréhensible. Par exemple, le *Mode autonome* de la *Ressource* `Manipulateur` utilise une *Action* `Déplacer Bras` qui permet de déplacer le bras mécanique dans son espace atteignable vers une position donnée. Cette *Action* a été définie à partir des *Commandes* rattachées à la *Ressource* `Manipulateur`.

Les *Perceptions* et *Générateurs d'événements* rattachés à une *Ressource* sont définis progressivement pendant chacune de ces étapes en fonction des besoins des *Modes*, *Actions* et *Commandes* et en fonction des capteurs qui peuvent être exploités. Les *Générateurs d'événements* et les *Perceptions* peuvent faire partie d'une *Ressource*, ou non, suivant le choix des développeurs. Nous proposons quelques règles permettant d'aider à faire ce choix :

1. Un *Générateur d'événements* (resp. une *Perception*) produisant des *événements* (resp. des *données d'état*) d'origine proprioceptive sur les *éléments matériels*, doit être agrégé dans une *Ressource* contrôlant ces *éléments matériels*. Par exemple, un *Générateur d'événements* notifiant l'approche ou la sortie d'une configuration singulière du bras mécanique, se base sur des données issues des capteurs de position angulaire embarqués sur le bras mécanique, il sera agrégé dans la *Ressource* `Manipulateur` contrôlant le bras mécanique.

2. Un *Générateur d'événements* (resp. une *Perception*) produisant des événements (resp. des *données d'état*) d'origine proprioceptive ne doit pas être agrégé dans une *Ressource* s'il se base sur des données provenant d'éléments matériels non contrôlés par une *Ressource*. Par exemple, un *Générateur d'événements* notifiant la baisse d'énergie ne sera pas agrégé dans une *Ressource* s'il n'existe pas de *Ressource* chargée de contrôler spécifiquement la consommation d'énergie du robot.
3. Un *Générateur d'événements* (resp. une *Perception*) produisant des *événements* (resp. des *données d'état*) d'origine extéroceptive partagés par l'ensemble des *Ressources* d'un contrôleur, ne doit pas être agrégé dans une *Ressource*. Par exemple, une *Perception* permettant de positionner le robot dans l'environnement à partir de mesures GPS, n'est pas agrégé dans une *Ressource*, puisque toutes les *Ressources* peuvent avoir besoin de cette donnée.
4. Un *Générateur d'événements* (resp. une *Perception*) produisant des *événements* (resp. des *données d'état*) d'origine extéroceptive doit être agrégé dans une *Ressource*, si celle-ci a pour tâche, entre autre, de fournir ces données au reste du contrôleur. Par exemple, pour une camera motorisée embarquée sur le véhicule, s'il existe une *Ressource* `Vision` qui permet de la contrôler, alors les *Perceptions* (traitement de l'image) sont agrégées par cette *Ressource*. Si la `Vision` n'existe pas, mais que le `Mobile` contrôle une caméra en plus du véhicule, ces *Perceptions* pourront être agrégées dans cette *Ressource*. Si, par les choix de conception des développeurs, la caméra ne fait partie d'aucune *Ressource* et que les données qu'elle propose sont partagées par l'ensemble des *Ressources*, alors la troisième règle devrait s'appliquer.

Evidemment, même avec ces quelques règles, le rattachement des *Générateurs d'événements* et des *Perceptions* aux *Ressources* reste essentiellement dépendant du point de vue des développeurs, et en particulier de la manière dont ils ont décomposé le contrôleur en différentes *Ressources*.

La phase d'analyse est cyclique (cf. fig. 6.3), chaque cycle permettant de détailler ou de remettre en cause les étapes des cycles précédents. En effet, les développeurs peuvent avoir besoin de nouveaux éléments matériels nécessaires à la réalisation de certains services ou peuvent avoir besoin de nouveaux services. L'ajout de nouveaux éléments matériels et de nouveaux services peut amener à reconsidérer les boucles de réactivité directe et plus généralement à reconsidérer les *Ressources* identifiées (modification des *Ressources* existantes ou création de nouvelles *Ressources*).

Durant la phase d'analyse, aucun outil ou notation n'est imposé par la méthodologie, chaque expert utilisant librement ceux qui lui conviennent. Par exemple, les automaticiens ont l'habitude de travailler avec des outils de modélisation mathématique, comme MatLab, qui sont bien adaptés aux problématiques posées par cette phase (e.g. modélisation des lois de commande). Toutefois, nous préconisons d'utiliser si possible des notations standards pour chaque domaine d'expertise et en premier lieu UML.

6.3 Modélisation UML

Le modèle d'organisation d'architecture est présenté par la suite de façon plus détaillée, sous la forme d'un ensemble de diagrammes UML. La modélisation UML donne la possibilité de formaliser la description précédente afin d'écrire des spécifications plus claires et compréhensibles.

6.3.1 Vue statique

La vue statique de l'architecture est présentée sous la forme d'un ensemble de diagrammes de classes UML. Le premier diagramme présenté dans la figure 6.4, est décrit à partir de classes abstraites qui représentent les deux grands types d'entités qu'il est possible d'identifier grâce à la méthode de

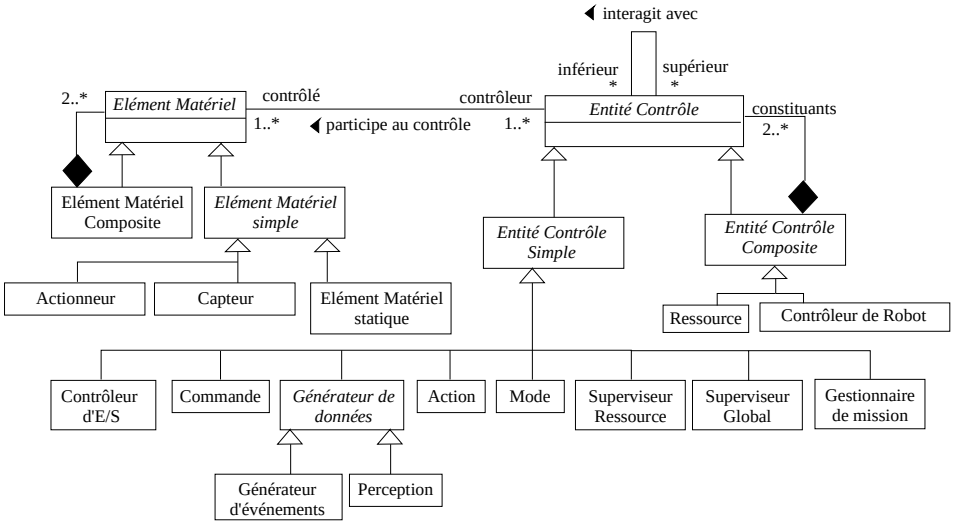


FIG. 6.4 – Vue statique (formalisme UML) - entités de contrôle et éléments matériels

conduite de l'analyse précédente. Il s'agit des *éléments matériels*, qui permettent de décrire la partie physique du robot, et les *entités de contrôle*, qui permettent de décrire le contrôle du robot.

Chaque *élément matériel* peut être lui-même un *élément simple* comme un *capteur*, un *actionneur* ou un *élément matériel statique* (e.g. un segment du bras mécanique), ou un *élément matériel composite* (e.g. le bras mécanique). Un *élément matériel* est soumis au contrôle d'un ensemble d'*entités de contrôle* (association *participe au contrôle*). Le concept d'*entité de contrôle* est une abstraction de l'ensemble des entités présentes dans le modèle d'architecture défini en début de chapitre. Ce concept regroupe celui de *Mode*, *Superviseur de Ressource*, *Action*, *Commande*, *Générateur d'événements*, *Perception*, *Superviseur Global*, *Gestionnaire de Mission* et *Contrôleur d'Entrées/Sorties*. Il regroupe également les entités qui apparaissent comme des composites dans cette architecture, à savoir les concepts de *Contrôleur de robot* et de *Ressource*. Chaque *entité de contrôle* est conceptuellement située dans une des couches de l'architecture logicielle, c'est pourquoi il existe une relation (traduite par l'association *interagit avec*) permettant de les ordonner entre elles.

Les deux diagrammes suivants représentent l'organisation systémique (fig. 6.5) et l'organisation hiérarchique (fig 6.6) des *entités de contrôle*. Ils précisent en particulier les cardinalités, non-exprimées dans le modèle informel. Le diagramme de la figure 6.5 formalise, par exemple, le fait qu'il y a un *Superviseur Global* et au plus un *Gestionnaire de mission*, pour un *Contrôleur de Robot* donné. De la même façon, on voit sur le diagramme de la figure 6.6 qu'un *Mode* ne peut interagir qu'avec un unique *Superviseur de Ressource*.

Ces deux diagrammes permettent de déduire quelles *entités de contrôle simples* peuvent avoir des services exportés par des *entités de contrôle composites*. Pour celà, il suffit de regarder les interactions entre *entités de contrôle simples* (fig. 6.6) et la façon dont ces entités sont réparties dans les *entités de contrôle composites*. Prenons l'exemple d'une *Ressource*, qui est une *entité de contrôle composite*. Elle exporte les services rendus par son *Superviseur de Ressource* car il y a interaction entre le *Superviseur Global* et le *Superviseur de Ressource*. Elle peut exporter les services de certains de ses *Générateurs de données*, car il peut y avoir une interaction directe entre un *Générateur de données* et le *Superviseur Global*. Une *Ressource* exporte les demandes de services associées aux entités *Com-*

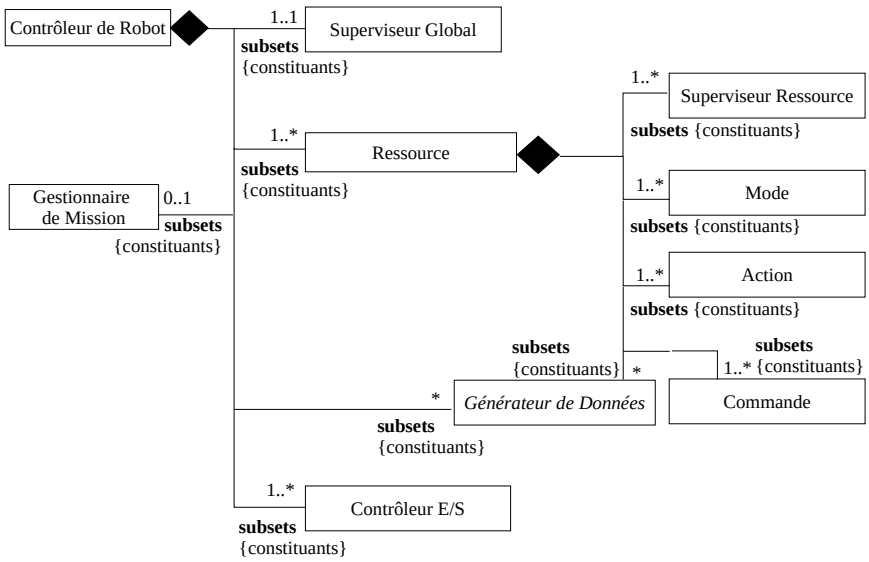


FIG. 6.5 – Vue statique : organisation systémique

mandes et *Générateurs de données* qu'elle agrège, afin que celles-ci interagissent avec les *Contrôleurs d'Entrées/Sorties* (e.g. une *Commande* envoie des *intentions* à un *Contrôleur d'Entrées/Sorties*). Il en va de même si on prend pour exemple le *Contrôleur de Robot*, qui exporte les services définis par son *Superviseur Global* car il y a une interaction directe entre le *Superviseur Global* et la *Plate-forme opérateur*.

Le diagramme de la figure 6.6 met l'accent sur les relations hiérarchiques entre *entités de contrôle simples*. Premièrement, il représente les relations hiérarchiques entre entités de couches directement inférieures et supérieures. Deuxièmement, il représente les relations entre entités de couches non-adjacentes : les *Générateurs de données* sont potentiellement reliés avec plusieurs *entités de contrôle simples* appartenant à des couches non directement supérieures ou inférieures. Troisièmement, il représente des relations hiérarchiques au sein d'une même couche : un *Mode de coopération* peut être en relation avec d'autres *Modes de coopération* d'autres *Ressources* ; un *Superviseur Global* est en relation avec un *Gestionnaire de Mission* qui se charge de la planification. Enfin, il représente la relation spécifique qui est induite par une téléopération directe : la *Plate-forme opérateur* est directement reliée avec une *Action* qui sert de "serveur de téléopération" (gestion des retards de communication).

Le diagramme de la figure 6.7 présente les données échangées pendant les interactions entre *entités de contrôle*. Ces données sont les *intentions* et les *observations*, en référence à la figure 6.1. Une *observation* est une information destinée aux *entités de contrôle* des couches supérieures. Chaque *entité de contrôle* est capable d'envoyer différentes *observations* aux entités des couches supérieures, qui prennent la forme de *données d'état*, de *comptes-rendus* ou d'*événements*. Un *événement* représente un phénomène sporadique. Une *donnée d'état* représente un état de la partie opérative du robot ou de son environnement (e.g. donnée capteur, carte de l'environnement). Un *compte-rendu* est une *observation* liée à l'exécution d'un *ordre* (échec, statut de l'exécution, etc.). Une *observation* peut être réalisée à partir d'informations disponibles sur un ensemble d'*éléments matériels*. Par exemple, la détection d'un *événement* lié à l'approche d'un obstacle se base sur des capteurs ultrasons placés à l'avant du véhicule, l'*élément matériel* sur lequel se base cette *observation* est donc le `véhicule`. Une *intention* est une information produite par une *entité de contrôle* destinée à influencer l'activité

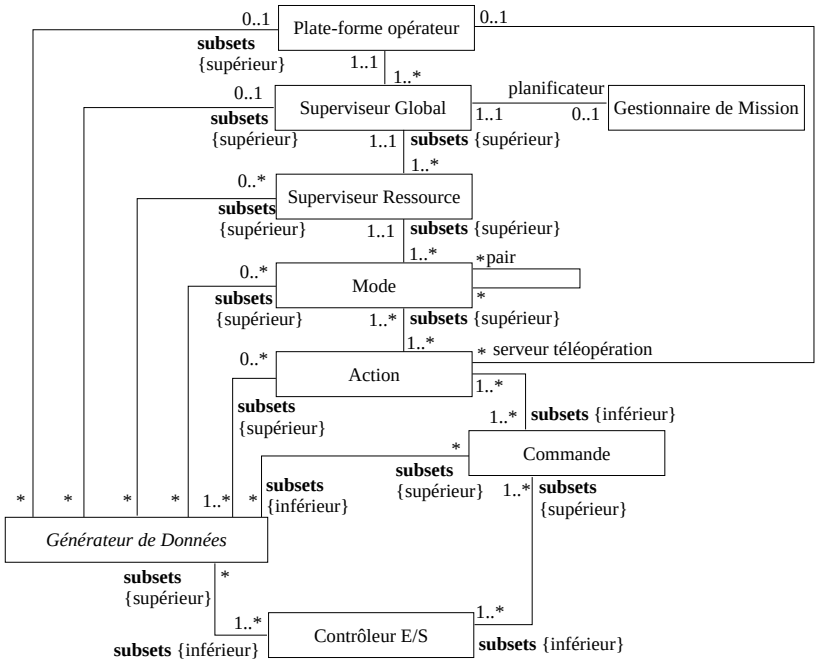


FIG. 6.6 – Vue statique : organisation hiérarchique des entités de contrôle simples

des *entités de contrôle* des couches inférieures. Chaque *intention* peut prendre différentes formes, comme une *ordre* (qui peut mener à la génération de *comptes-rendus*), une *demande de génération* (qui mène explicitement à la génération d'*événements* ou de *données d'état*), une *consigne* (envoyée aux *Commandes*), un *vecteur de données de commande* (envoyé aux *Contrôleurs d'Entrées/Sorties*). Une *intention* peut concerner, directement ou non, le contrôle d'un ensemble d'*éléments matériels* donné. Par exemple, une demande de réalisation d'un *ordre* de déplacement du véhicule concerne l'*élément matériel* `Véhicule` considéré.

Une *entité de contrôle* intègre une ou plusieurs *Stratégies de décision* sur lesquelles elle se base pour définir les *intentions* et *observations* émises. Une *Stratégie de décision* est typiquement une *Loi de commande* pour les *Commandes* et plusieurs pour les *Actions* (chargées de la commutation et du paramétrage des différentes *Commandes*). Une *Stratégie de décision* est une *Stratégie d'observation* pour les *Générateurs d'événements* et *Perception*. Il est évidemment possible de définir d'autres *Stratégies de décision* qui ne sont pas directement basées sur des *éléments matériels* spécifiques, par exemple des *Stratégies de planification* pour certains *Modes* et le *Gestionnaire de Mission*.

La vue statique du modèle d'architecture générique proposé ayant été décrite, nous présentons dans la section suivante sa vue dynamique, c'est-à-dire les interactions entre les *entités de contrôle* présentées dans ces diagrammes.

6.3.2 Vue dynamique

Les diagrammes présentés dans cette section illustrent les interactions typiques entre des objets issus des classes décrites dans la vue statique. Ces diagrammes aident les développeurs à mieux saisir les interactions potentielles entre les *entités de contrôle*.

Le premier diagramme (cf. fig. 6.8) illustre les interactions entre les *entités de contrôle composites*, déclenchées par l'envoi d'une demande de réalisation d'*ordre* par un opérateur humain. L'opérateur

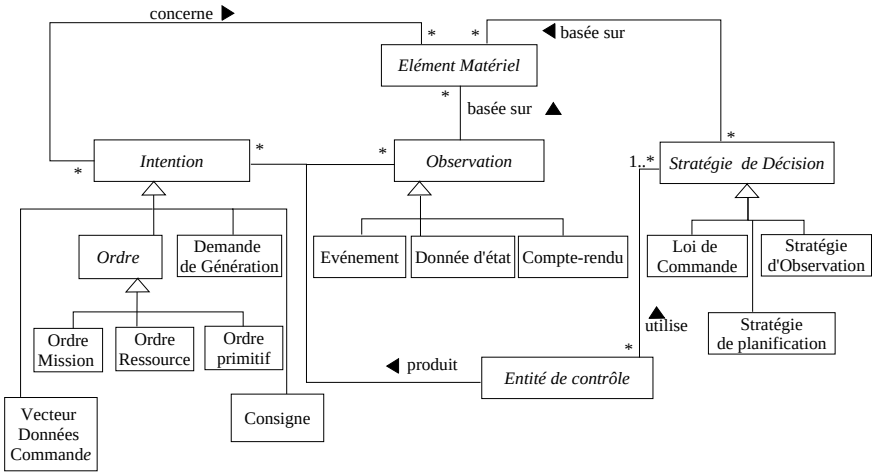


FIG. 6.7 – Vue statique : Décision, intentions et observations

envoie un message `exécuter(Ordre de Mission)` au *contrôleur de robot*, qui délègue les appels de services qu'il reçoit au *Superviseur Global* qu'il contient. Dans l'exemple présenté, ce *Superviseur Global* ne peut directement traduire l'*ordre de mission* en un *ordre de ressource*. De fait, il demande au *Gestionnaire de Mission* de réaliser la planification des *ordres de ressource* à réaliser afin de réaliser l'*ordre de mission*. Le *Gestionnaire de Mission* renvoie le plan de mission au *Superviseur Global* afin que celui-ci supervise sa réalisation. Pour ce faire, il sélectionne les *Ressources* adéquates à la réalisation de chaque *ordre de ressource* contenu dans le plan de mission et leur demande de les réaliser. Si dans l'exemple présenté l'engagement des *Ressources* dans la réalisation des *ordres de ressource* est présenté de manière séquentielle, rien n'empêche que cet engagement se fasse en parallèle. Les *Ressources* comme le *Superviseur Global* envoient des *comptes-rendus* de la réalisation des *ordres* qu'ils ont reçus (respectivement des *ordres de ressource* et des *ordres de mission*). Ces *comptes-rendus* peuvent contenir des informations relatives soit au succès de la réalisation d'un *ordre*, soit à l'échec ou à l'impossibilité de le réaliser.

Dans le cas d'un échec, le *Superviseur Global* sera dans la plupart des cas amené à redemander la génération d'un plan par le *Gestionnaire de mission* (cas non-représenté dans la figure 6.8). L'échec en lui-même peut avoir deux origines différentes. Il peut provenir directement d'une des *Ressources* engagées, qui renvoie une situation d'échec à travers un *compte-rendu* ou qui notifie un *événement* (local à la *Ressource*) dont le *Superviseur Global* a demandé la génération. L'échec peut aussi provenir d'un *Générateur d'événements* qui notifie le *Superviseur Global* d'un *événement* considéré comme global pour l'intégralité du robot (i.e. quelles que soient les *Ressources* engagées à un moment donné). Ce deuxième cas est présenté dans la figure 6.9 : le *Superviseur Global* active, avant l'exécution d'un *ordre de ressource*, un *Générateur d'événements* qui a pour but de détecter et notifier des *événements* globaux (cf. `demande Génération Evénement Global`). Les *événements* globaux sont ceux qui ne sont pas relatifs à une *Ressource* du contrôleur. L'identification de ces *événements* relève donc essentiellement de la façon dont les développeurs ont décomposé un contrôleur en terme de *Ressources*. Par exemple, dans le cas du robot manipulateur mobile, l'*événement* `Batterie Faible` est considéré comme global, puisqu'aucune *Ressource* ne contrôle la batterie. La figure montre qu'à certains moments de l'exécution d'un *ordre de ressource*, un *événement* global peut être notifié au *Superviseur Global*, qui décide alors d'annuler la réalisation de l'*ordre de ressource* courant (envoi d'un *ordre* d'arrêt à la *Ressource* concernée). Ce scénario prend place lorsque le *Superviseur Global*

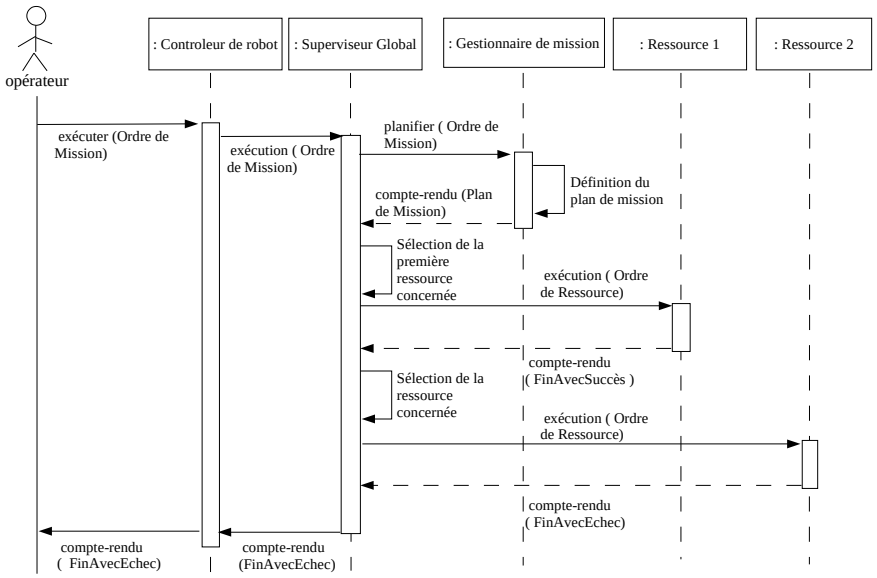


FIG. 6.8 – Vue dynamique : réalisation de missions

reçoit un *événement* `Batterie Faible` et qu'il décide d'arrêter la mission car le robot ne dispose plus de l'énergie nécessaire.

Chaque *Ressource* réalise de façon interne l'ensemble des *ordres de ressource* qu'elle reçoit du *Superviseur Global*. Un exemple est présenté dans la figure 6.10. En fonction de l'*ordre de ressource* à exécuter, le *Superviseur de Ressource* sélectionne le *Mode* adéquat à sa réalisation. Par exemple, pour le contrôle du véhicule du robot manipulateur mobile, il existe une *Ressource* `Mobile` qui peut soit être téléopérée à distance (`Mode téléopération`), soit être autonome pour la réalisation de différents *ordres de ressource* (`Mode autonome`). Dans le cas où l'*ordre* reçu par la *Ressource* est une demande de téléopération, c'est le `Mode téléopéré` qui sera engagé par le *Superviseur de Ressource* du `Mobile`. Chaque *Mode* procède, si besoin est (en particulier dans le cas du *Mode autonome*), à une planification locale de l'*ordre* reçu par la *Ressource*. Il traduit l'*ordre de ressource* reçu en une succession d'*ordres primitifs*. Une fois cette planification locale effectuée, un *Mode* réalise séquentiellement ces *ordres primitifs* en utilisant les *Actions* adéquates (cf. `exécuter(Ordre Primitif)`). De la même façon que pour le *Superviseur Global*, un *Mode* peut avoir besoin d'être notifié d'*événements* qui peuvent remettrent en cause la séquence d'*ordres primitifs* initialement planifiée. Pour cela, le *Mode* demande à un ou plusieurs *Générateurs d'événements* d'être notifié de ces *événements* (cf. `exécuter(Demande Génération)`), en fonction de l'*Action* courante. Par exemple, pour le `Mode téléopéré` du `Mobile`, il existe une *Action* `VéhiculePiloterParOpérateur` qui permet à l'opérateur de piloter directement le véhicule. Néanmoins, rien n'empêche que le véhicule rencontre un obtacle ; c'est pourquoi le `Mode téléopéré` demande à un *Générateur d'événements* de lui notifier la présence d'obstacles. Si le *Générateur d'événements* notifie la présence d'un obstacle, le `Mode téléopéré` peut, par exemple, arrêter l'*Action* courante, puis démarrer une *Action* qui va permettre au véhicule d'éviter automatiquement l'obstacle. Une fois l'obstacle franchi (un *événement* notifie le franchissement), le `Mode téléopéré` peut redémarrer l'*Action* initiale afin que l'opérateur puisse reprendre la main sur le pilotage du véhicule. Dans le cas d'un `Mode autonome` l'interruption d'une *Action* peut amener à une nouvelle planification des *ordres primitifs* à réaliser (cf. fig 6.10).

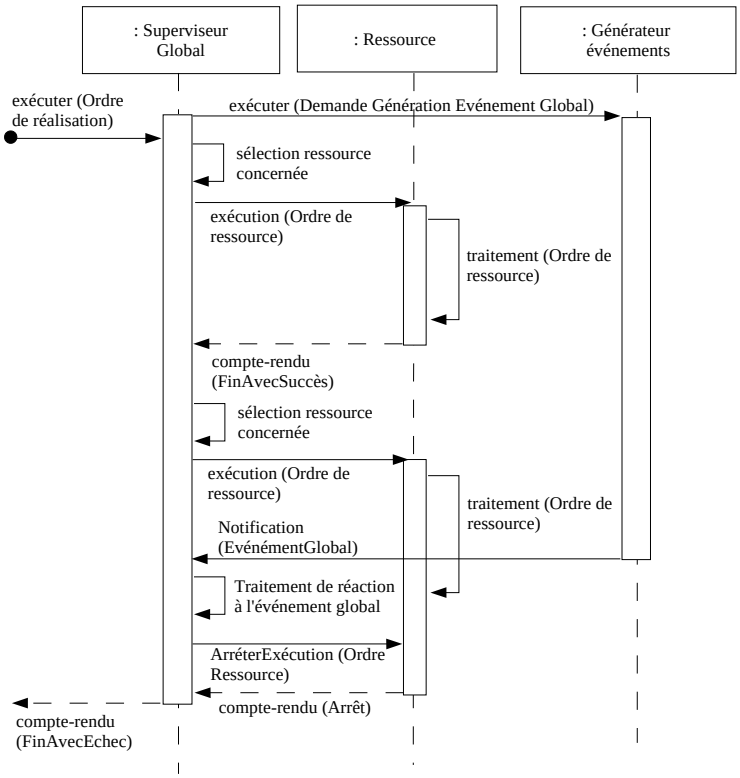


FIG. 6.9 – Vue dynamique : activation des Ressources

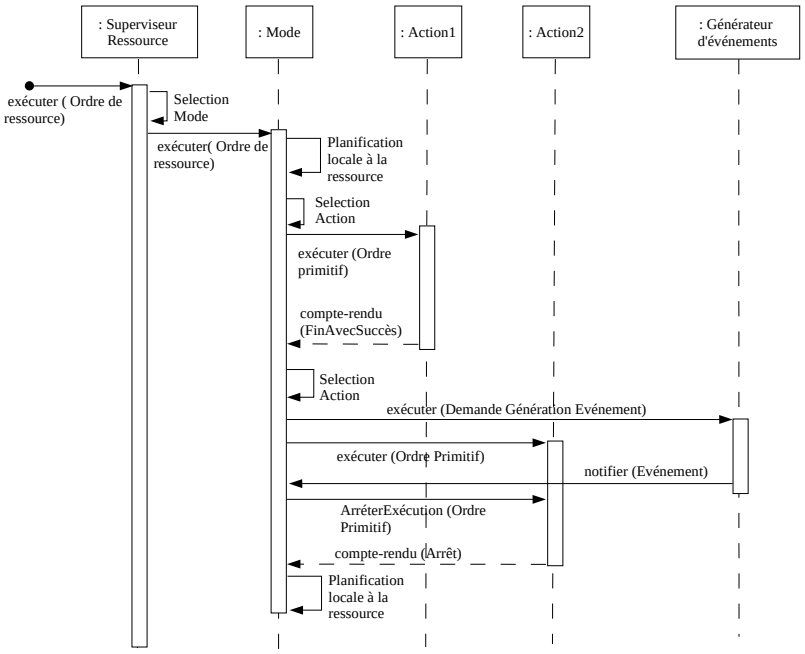


FIG. 6.10 – Vue dynamique : enchaînement d'actions

Chaque *Action* permet de réaliser un *ordre primitif*, elle utilise pour cela un ensemble de *Commandes*. La figure 6.11 présente les interactions typiques entre une *Action* et un ensemble de *Commandes*. Nous limitons notre réflexion sur les interactions entre *Actions* et *Commandes* au cadre de la commutation de la *Commande* active, sans considérer celui de l'activation de *Commandes* concurrentes. Chaque *Commande* contrôle l'application d'une *loi de commande* spécifique qui possède un domaine de validité propre. L'*Action* utilisant une *Commande*, elle doit prendre en compte ce domaine de validité afin de savoir quand et comment réaliser la commutation. Par exemple, lorsque nous utilisons certaines *Lois de commande* permettant de contrôler le bras mécanique du robot manipulateur mobile, il faut prendre en compte le fait qu'elles peuvent être sensibles à des configurations singulières. Il est alors nécessaire de commuter la *Commande* active vers une *Commande* qui applique une *Loi de commande* insensible à ces singularités (qui peuvent mettre en péril le contrôle du bras mécanique). Ainsi, chaque *Action* possède potentiellement un schéma de commutation qui lui permet, en fonction des *événements* dont elle est notifiée, d'activer, de désactiver et de synchroniser les *Commandes* avec lesquelles elle interagit. Ce mécanisme, présenté dans la figure 6.11, est le même que celui utilisé pour l'enchaînement d'*Actions* par un *Mode*. Il se base sur un ensemble de *Générateurs d'événements* activés en fonction du contexte d'exécution de l'*Action*, permettant de lui notifier les *événements* à partir desquels elle prendra la décision de commutation. Dans l'exemple du contrôle du bras mécanique, il existe un `GénérateurEvénementsSingularités` permettant aux *actions* du `Manipulateur` de décider des commutations. Notons que pour prendre les décisions de commutation, l'*Action* doit manipuler l'ensemble des *lois de commande* appliquées par les *Commandes* avec lesquelles elle interagit.

Le dernier diagramme (cf. fig. 6.12) présente les interactions entre les entités situées dans les couches les plus basses d'une architecture. Ces interactions sont, le plus souvent, basées sur un schéma périodique. A partir du moment où une *Commande* a été démarrée, son comportement

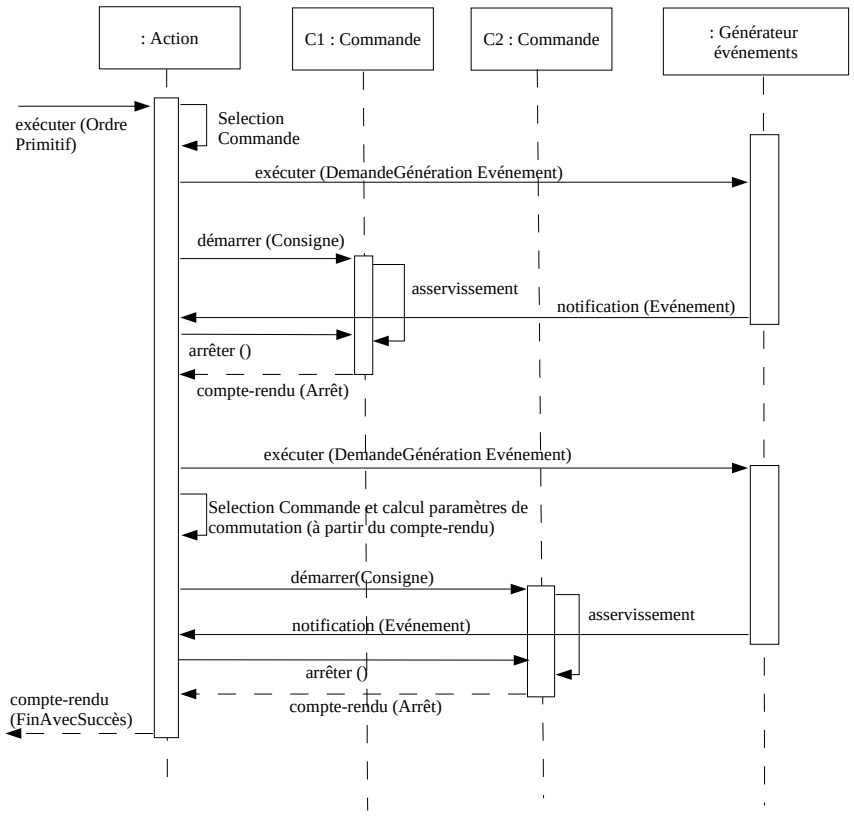


FIG. 6.11 – Vue dynamique : commutation de commandes

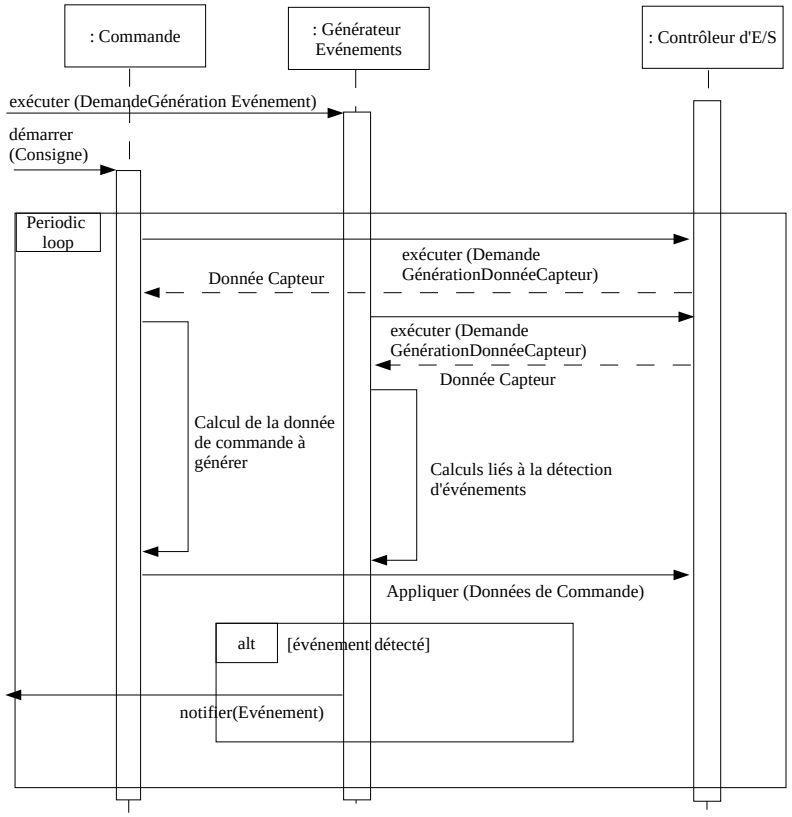


FIG. 6.12 – Vue dynamique : interactions périodiques

devient périodique et ses interactions avec le *Contrôleur d'Entrées/Sorties* le deviennent également. A chaque période, une *Commande* demande aux *Contrôleurs d'Entrées/Sorties* de lui fournir les données capteurs mises à jour, calcule les *vecteurs de données de commande* à appliquer aux actionneurs, puis fournit aux *Contrôleurs d'Entrées/Sorties* ces données. De la même façon, un *Générateur de données* demande aux *Contrôleurs d'Entrées/Sorties* de lui fournir les données capteurs mise à jour. Un *Générateur d'événements* effectue éventuellement des calculs intermédiaires sur ces données capteurs, avant de passer à une étape de détection (e.g. un seuillage) qui peut mener à la notification d'un *événement* (cf. le bloc conditionnel `alt`, fig. 6.12).

Ces diagrammes nous permettent d'identifier différentes interactions "typiques" dans le modèle d'architecture proposé :

- des interactions de type "requête/réponse" autour des services de réalisation d'*ordres* provenant du *Superviseur Global*, des *Superviseurs de Ressources*, des *Mode* et *Actions*. Une entité envoie une requête (associée à un *ordre*) à une entité d'une couche inférieure qui réalise cet *ordre* et renvoie en fin d'exécution un *compte-rendu* (échec ou succès). Ce type d'interaction peut être décliné de différentes façons. Par exemple, le receveur de la requête peut envoyer des *comptes-rendus* pendant son exécution (e.g. acquittement de réception de la requête) ou peut recevoir des demandes d'interruption.
- des interactions de type "enregistrement/notification" autour des services de génération de *données d'état* et d'*événements*. Ces interactions peuvent prendre place entre n'importe quelles entités d'une architecture : une entité demande à une entité d'une couche inférieure de s'abonner

à une *donnée d'état* ou un *événement* que celle-ci génère, l'entité de couche inférieure notifie l'entité supérieure dès que l'*événement* est détecté ou que la *donnée d'état* est mise à jour. Ce type d'interaction peut être décliné de différentes façons. Par exemple, une *Perception* génère des *données d'état* tant que l'entité abonnée n'a pas demandé explicitement à se désabonner. Un *Générateur d'événements* va, à l'inverse, arrêter la notification dès que l'*événement* est détecté.

– des interactions de type "exécuter/arrêter" autour du contrôle de l'activité des *Commandes* par les *Actions*. Les *Actions* demandent explicitement le démarrage et l'arrêt de *Commandes* qui renvoient, après arrêt, des *données d'états*. Ce type d'interaction peut être décliné de différentes façons, par exemple en laissant la possibilité aux *Actions* de configurer différents paramètres des *Commandes*. Il diffère du type d'interactions "requête/réponse" en ce sens qu'il ne peut pas y avoir de terminaison automatique des *Commandes*, mais que l'arrêt de leur exécution doit être explicitement demandé par l'*Action*.

6.4 Conclusion

A partir du modèle d'organisation d'architectures et de la démarche d'analyse proposés dans ce chapitre, les développeurs possèdent les concepts permettant de décomposer leurs contrôleurs en un ensemble d'entités représentant des concepts *métiers*. Ce modèle a l'avantage d'être découplé de toute solution technologique ou notation utilisée pour sa mise en œuvre. Il repose sur un ensemble de diagrammes qui permettent de montrer précisément la façon dont sont organisées les entités de contrôle. Ce modèle est innovant dans le sens où il propose une formalisation des *responsabilités* des entités de contrôle, à partir desquelles un développeur peut identifier les entités de contrôle et interactions nécessaires à la description d'une architecture de contrôle pour un robot donné. L'utilisation d'UML s'est révélée bien adaptée à la formalisation et à la documentation du modèle d'organisation. Ceci nous amène à penser que l'utilisation généralisée d'un langage de modélisation standard, permettrait de mieux comprendre et échanger les principes d'organisation proposés dans les travaux sur les architectures logicielles des contrôleurs. En ce sens, notre proposition est une base exemplaire pour les propositions futures de modèles d'architecture.

Le modèle proposé synthétise certains types d'interactions inhérents aux architectures de contrôle mixtes mais il n'intègre pas, à l'heure actuelle, les types d'interactions induits par des mécanismes d'arbitrage, tel que ceux utilisés par les architectures comportementales. Nous pensons qu'intégrer, dans ce modèle, ces types d'interactions spécifiques, permettrait de tirer complètement parti des deux modèles d'organisation "classiques" (délibératif et comportemental). Le but à long terme serait de définir une démarche d'analyse/conception consensuelle.

Le chapitre suivant présente le langage à composants CoSARC, qui va permettre de poursuivre l'ensemble des phases de conception, de programmation, de déploiement et d'exécution au sein du processus défini par la méthodologie CoSARC.

Résumé :

Un modèle d'organisation d'architectures de contrôle :

- propose une organisation hiérarchique et systémique d'une architecture contrôle.

- définit la mise en relation directe de couches non-adjacentes (approche « mixte »).

- définit les responsabilités assumées par des entités de contrôle et par des (sous-)systèmes, ainsi que leurs interactions typiques, autour du concept de **Ressource**.

- constitue le support de la phase d'analyse.

Chapitre 7

Langage à composants CoSARC

7.1 Généralités

Ce chapitre présente la définition du langage CoSARC, langage à composants qui permet de modéliser et de programmer des architectures logicielles à base de composants réutilisables. Cette première section présente les principes d'utilisation du langage ainsi que ses concepts généraux. Le langage s'inspire des différentes approches à composants existantes et met en adéquation les notations utilisées et les besoins constatés pour le développement d'architectures logicielles de contrôle. Ces notations sont présentées à travers un meta-modèle.

7.1.1 Définitions et Concepts

Cette section regroupe la définition des concepts génériques gravitant autour de la notion de composant et de langage de composants. L'ensemble de ces concepts, est représenté dans le meta-modèle du langage CoSARC, exposé dans les figures 7.1, 7.2 et 7.3. Les lecteurs peuvent se référer à la spécification du meta-modèle d'UML 2.0 [65] qui a inspiré la définition de ce meta-modèle.

Définition 1 *Langage à composants.*

Un langage à composants est un langage permettant de décrire des composants logiciels et des architectures logicielles issues de leur assemblage. Les programmes écrits dans ce langage sont définis, se déploient et s'exécutent dans un environnement logiciel de composants. Un environnement logiciel est constitué de différents outils logiciels permettant de manipuler des composants et des architectures à tout moment de leur cycle de vie (modélisation, programmation, déploiement, exécution).

Cette première définition, très large, nécessite la définition des concepts manipulés dans le langage. Ces concepts sont représentés dans les figures 7.1 et 7.2.

Définition 2 *Descripteur de composants.*

Un descripteur de composants est une entité informatique qui permet de créer un ensemble de composants instances et qui définit un type pour ces composants. Il possède des propriétés internes qui sont, par exemple, des attributs et des opérations. Il contient également des descripteurs de port qui définissent les points de connexion des composants instances. Un descripteur peut spécialiser un unique autre descripteur ; dans ce cas le descripteur fils doit posséder des descripteurs de port qui référencent les mêmes interfaces que les descripteurs de port de son père et peut également ajouter de nouveaux descripteurs de port fournis.

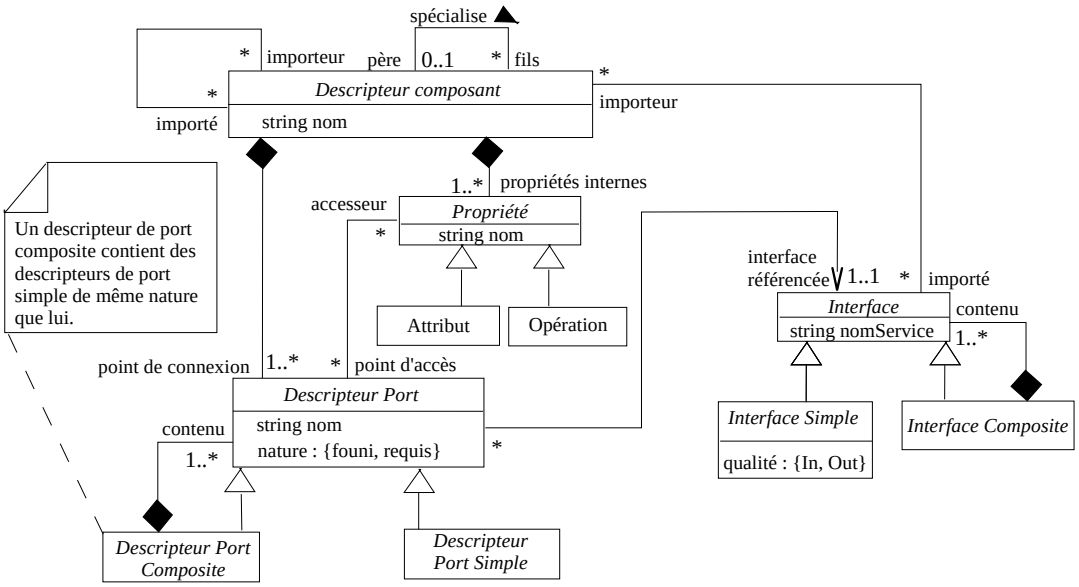


FIG. 7.1 – Meta-modèle des descripteurs de composants (formalisme UML)

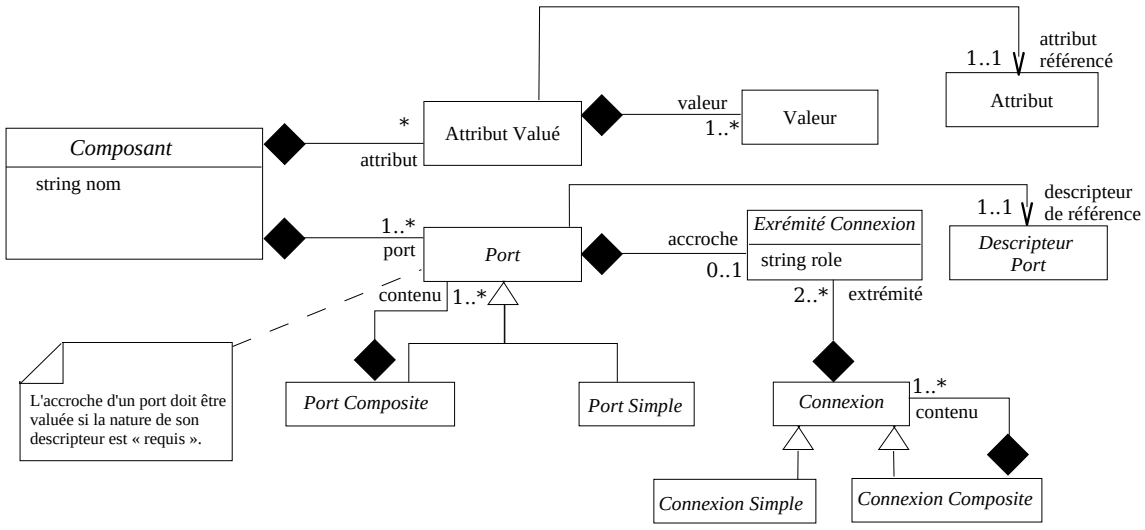


FIG. 7.2 – Meta-modèle des composants (UML)

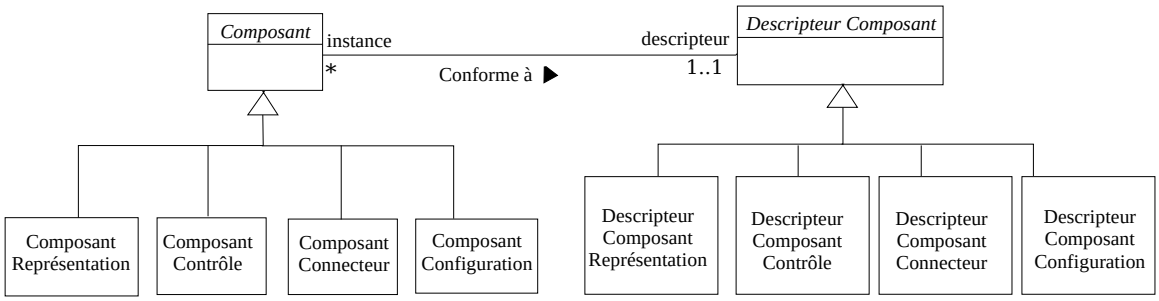


FIG. 7.3 – Meta-modèle de la relation composant-descripteur de composants

Définition 3 *Composant.*

Un composant est une entité informatique, instance d'un descripteur de composant, qui est exécutée dans un environnement d'exécution du langage CoSARC. Un composant est fait de ports, d'attributs, et d'opérations. Ses ports sont des instances des descripteurs de ports définis dans le descripteur dont il est instance. Ses attributs sont ceux définis au niveau du descripteur dont il est instance, mais doivent tous être valués. Ses opérations sont celles définies dans le descripteur dont il est instance.

Le langage CoSARC propose deux mécanismes de composition différents pour les composants : l'assemblage et l'agrégation. L'assemblage consiste à créer des connexions entre des ports de composants. Les concepts suivants participent à la définition du mécanisme d'assemblage.

Définition 4 *Interface*

Une interface est un contrat qui définit la manière dont un composant offre ou utilise un service (i.e. une fonctionnalité). Le nom de l'interface correspond au nom donné au service offert ou utilisé. Une interface peut-être simple ou composite.

Définition 5 *Descripteur de port.*

Un descripteur de port est une entité qui contient la description d'un ou plusieurs ports. Un descripteur de port peut être simple ou composite. Il référence une interface qui déclare la façon dont un service est offert ou utilisé à travers ses ports instances. Il définit également si ses ports instances sont requis ou fournis, c'est-à-dire s'ils doivent respectivement être obligatoirement connectés ou pas.

Définition 6 *Port.*

Un port d'un composant est un point de connexion entre ce composant et un ou plusieurs autres composants. Un port est instance d'un descripteur de port, il est donc simple ou composite en fonction de la nature de son descripteur. Un port est fourni s'il permet d'utiliser un service implanté par le composant et il n'est pas obligatoirement connecté. Un port est requis s'il exprime le besoin qu'a un composant d'utiliser un service implanté par un ou plusieurs autres composants. Un port requis est obligatoirement connecté. Un port est "typé" par l'interface référencée par le descripteur dont il est instance.

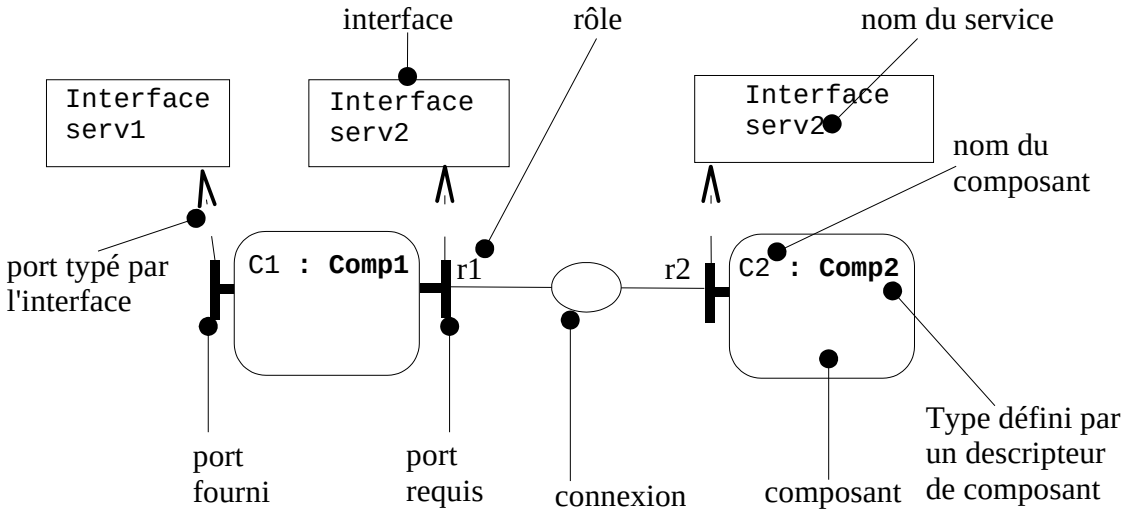


FIG. 7.4 – Diagramme d'un assemblage de composants (notation propre au langage CoSARC)

Définition 7 *Connexion.*

Une connexion est une entité qui permet de connecter deux ou plusieurs ports de composants. Lorsqu'une connexion est créée, les composants ainsi assemblés interagissent via leurs ports connectés. Une connexion possède au moins deux extrémités connectées à des ports. Une connexion est établie entre au moins un port requis et au moins un port fourni. Chaque connexion est instance d'un type de connexion particulier (cf. fig. 7.5) et prédéfini. A chaque extrémité d'une connexion est associé un rôle, qui définit la façon dont le composant connecté interagit à travers la connexion.

L'assemblage des composants est réalisé en connectant leurs ports à l'aide d'entités appelées connexions : lorsqu'il est connecté, un port possède une accroche sur une extrémité de la connexion qui le connecte (cf. fig. 7.2). Les interfaces permettent d'établir la possibilité d'une connexion et permettent de vérifier la validité d'une connexion. Cette vérification se fait, en premier lieu, sur le nom des services : si deux interfaces possèdent le même nom de service, alors une connexion peut être établie entre deux ports (un fourni et l'autre requis) typés respectivement par l'une et l'autre de ces interfaces. Dans le cas où les ports connectés sont composites, la vérification se fait récursivement sur les connexions établies entres les ports contenus. Ainsi, une connexion peut être elle-même composite, si elle est établie entre des ports composites. Le nombre de niveaux de composition des ports est conceptuellement infini dans le meta-modèle présenté jusqu'à présent, mais il est limité à deux dans les extensions du meta-modèle présenté par la suite, dans un souci de simplicité. La figure 7.4 présente la syntaxe graphique utilisée pour représenter les composants, connexion et interfaces.

Le langage CoSARC étant typé, nous avons défini dans le meta-modèle les concepts de *type* et de *variable* (cf. fig. 7.5). Il existe différents types possbiles pour une variable : un *Type de composants*, un *Type primitif* ou un *Types de connexion*. Chaque descripteur de composant définit un *Type de composant* qui possède le même nom que lui, il s'agit donc d'un type contruit par l'utilisateur. La hiérarchie des types de connexion manipulable est prédéfinie, mais elle n'est pas présentée pour des raisons de place. Il est facile de la reconstruire : il existe un type concret étendant le type abstrait *Type Connexion* pour chaque connexion spécifique (spécialisant *connexion*) présentée dans les figures

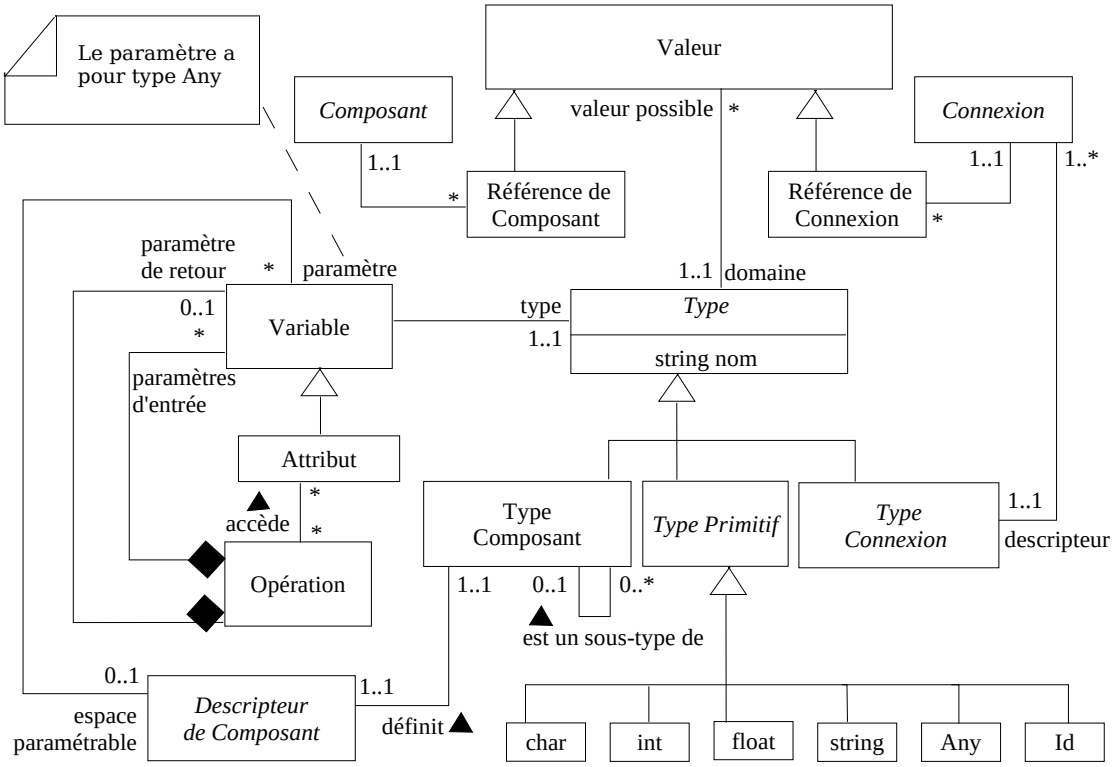


FIG. 7.5 – Meta-modèle du langage CoSARC : agrégation via les variables (formalisme UML)

suivantes. Les types primitifs sont des types classiques (int, string, etc.) en plus de deux types spécifiques : *Id* et *Any*. Une variable de type *Id* encapsule des informations d'identification d'un composant. Une variable de type *Any* a un type considéré comme indéfini jusqu'à l'instanciation (moment où leur type est fixé) d'un descripteur de composant dont elle est un des paramètres. En tant que paramètre d'un descripteur, elle a un nom unique qui peut être utilisé dans différents éléments de ce descripteur. Le type *Any* est une sorte de type "joker" qui nous évite d'alourdir le meta-modèle avec la gestion de la généricité paramétrique (i.e. templates) et qui couvre nos besoins de modélisation (cf. section sur les composants connecteurs).

Le mécanisme d'agrégation consiste à agréger des composants (assemblés ou non) à l'intérieur d'autres composants qui sont alors nommés *composites*. Un *composite* est l'unique responsable de l'assemblage des composants qu'il agrège : l'assemblage ne peut donc être le fait d'aucun des composants participant à cet assemblage, il est le résultat de la création d'une connexion entre leur ports par le *composite* qui les agrège. Un *composite* délègue tout ou partie des services qu'il fournit, aux composants qu'il agrège, en utilisant leurs services fournis. Par défaut, n'importe quel composant peut-être un *composite*, dès lors qu'il a pour attributs un ensemble de composants (et de connexions). En effet un attribut d'un composant est une variable qui peut être typée par un *type de composant* ou par un *type de connexion* (prédéfini). Un tel attribut a alors pour valeur une référence soit sur un composant, soit sur une connexion (cf. fig. 7.5). Un *composite* peut posséder des opérations dont les paramètres ont des types définis par des descripteurs de composants ou des types de connexion. Les opérations d'un *composite* peuvent contenir du code permettant de diriger l'ajout, le retrait, la substitution (remplacement d'un composant agrégé par un composant de même type ou d'un sous-type compatible) et l'assemblage (connexions des ports) de composants agrégés. Les opérations d'initia-

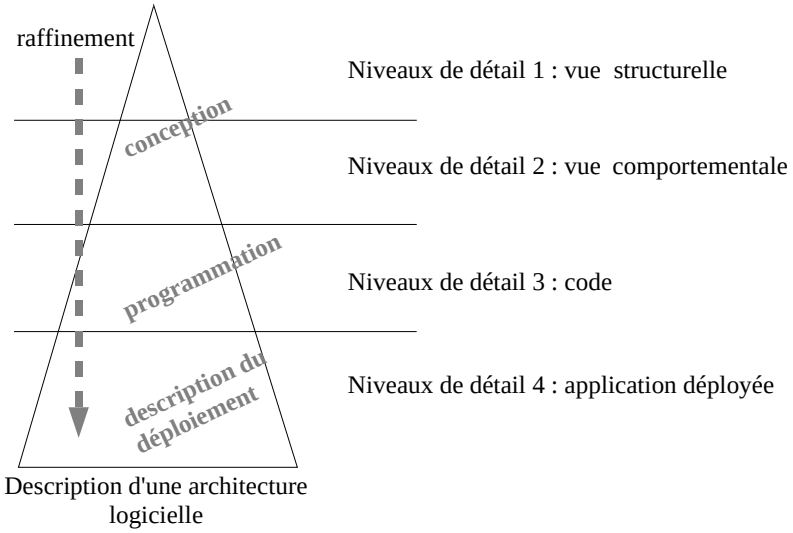


FIG. 7.6 – Pyramide des niveaux de détails

lisation et de destruction d'un *composite* permettent respectivement d'instancier et de détruire les composants agrégés et de les assembler de manière à créer l'architecture logicielle.

7.1.2 Raffinement

Actuellement, la phase de conception d'une application est documentée via un ensemble de diagrammes UML qui définissent des vues (structurelles et comportementales) sur l'application. Lorsqu'on passe à la phase de programmation, ces diagrammes sont traduits en code logiciel, c'est-à-dire dans un modèle exécutable. Un des problèmes d'utilisation d'UML est que rien ne garantit la correspondance entre les spécifications contenues dans les diagrammes et leur traduction sous la forme de code. Il existe des différences plus ou moins importantes entre les spécifications et le programme correspondant : le passage à un langage de programmation amène l'utilisateur à modifier, dans des proportions plus ou moins grandes, la structure des classes et leurs relations. D'autre part, la traduction des vues dynamiques se fait directement dans le code des opérations des objets, ce qui change complètement la manière de décrire ces vues. On peut alors se demander dans quelle mesure l'application programmée correspond encore à ses spécifications. Dans un souci de simplicité d'utilisation, nous avons défini un mécanisme de raffinement très simple : il est basé sur un pyramide de niveaux de détails (cf. fig. 7.6).

Définition 8 *Niveaux de détail et raffinement.*

Composants et architectures sont vus et manipulés à différents niveaux de détail en fonction de la phase du processus de développement. Pour un niveau de détail donné, on accède à un ensemble d'éléments de description, qui sont définis au niveau du meta-modèle. Un niveau de détail est dit supérieur à un autre lorsqu'il intègre tous les éléments de description de niveau inférieur et en ajoute de nouveaux. En terme d'abstraction, le niveau de détail inférieur est plus abstrait que le niveau de détail supérieur, car il occulte une partie des éléments de description utilisés. Le mécanisme d'ajout d'informations de description, pour passer à un niveau de détail supérieur, se nomme raffinement.

Le mécanisme de raffinement du langage est totalement dédié et donc bien moins générique que celui mis en œuvre via la transformation de modèles. Il permet à un descripteur de composant (et par conséquent à ses composants instances) d'évoluer durant tout le cycle de vie de l'application (conception, programmation, déploiement). Les développeurs peuvent manipuler un descripteur de composant au niveau de détail qui leur convient, en fonction de la phase dans laquelle ils se situent. Pour mettre en œuvre ce mécanisme de raffinement de façon simple nous partons du postulat qu'il existe une description globale d'un composant (resp. d'un descripteur de composant). Les développeurs peuvent manipuler certains éléments de cette description globale et n'ont pas accès à d'autres, en fonction du niveau de détail auquel ils se situent. Plus le niveau de détail est élevé et plus ils ont accès à de nombreux éléments de description. Le niveau de détail 1 est le moins élevé : les développeurs manipulent uniquement les descripteurs de composant sous leur forme structurelle (vision "boîte noire"). Le niveau de détail 2 détaille la vue comportementale des descripteurs de composants, par exemple en définissant les signatures de leurs opérations et leurs attributs, et/ou en utilisant des formalismes de description du comportement (e.g. machines à état). Le niveau de détail 3 consiste à écrire le code d'implantation des comportements (code des opérations et/ou en rajout d'information sur le modèle comportemental afin de le rendre exécutable). Le niveau de détail 4 est le plus élevé : les développeurs manipulent tous les éléments de description disponibles, y compris les détails liés au déploiement.

Ce processus de raffinement basé sur une description globale, s'il est très simple, apporte plusieurs avantages :

- Lorsqu'une information de description est ajoutée à un niveau de détail donné via un élément de description donné, la modification est répercutée immédiatement sur la description globale du composant. Les niveaux de détail dans lesquels cet élément de description est manipulable permettent aux développeurs de voir cette modification. La cohérence entre les niveaux de détail est ainsi directement assurée.
- La transition et la traçabilité entre les phases de conception, de programmation et de déploiement est facilitée, puisque le raffinement consiste simplement à rajouter des informations, en fonction des éléments de description disponibles pour un niveau de détail donné.

Evidemment, cela suppose que les notations utilisées pour écrire les descripteurs de composants sont adaptées à ce mécanisme de raffinement, c'est-à-dire qu'elles proposent des éléments de description adaptés à chaque niveau de détail et que ces éléments s'intègrent les uns avec les autres au sein d'une description globale. Au niveau de détail 1, ces notations doivent permettre de décrire la vue structurelle (les ports et interfaces). Au niveau de détail 2, elles doivent permettre de décrire la vue comportementale (signatures des opérations, diagramme d'état/transition) et de rattacher cette vue à la vue structurelle. Au niveau de détail 3, la vue comportementale doit pouvoir être enrichie avec du code (code des opérations). Au niveau de détail 4, la description est enrichie pour prendre en compte le contexte d'exécution, matériel et système, des composants.

Cette pyramide de niveaux de détail et le mécanisme de raffinement ont pour but de montrer la façon dont nous voulons utiliser le langage CoSARC.

7.1.3 Choix du formalisme de description de la vue comportementale

Pour permettre le raffinement du niveau de détail 2 vers le niveau de détail 3, une notation adaptée est nécessaire. En robotique, l'utilisation de notations traduisant les états et changements d'états des entités de contrôle et/ou de leurs interactions est très fréquent. Dans plusieurs notations utilisées à grande échelle, comme les diagrammes de machine d'état d'UML, le raffinement vers le code d'implantation de ces modèles se fait souvent de manière ad hoc, ce qui a plusieurs désavantages :

- Premièrement, rien n'assure que le code d'implantation corresponde au comportement spécifié sur le modèle formel.

- Deuxièmement, l'effort consenti par les programmeurs est grand. Le modèle ne sert que de "document de spécification" sur lequel ils s'appuient pour comprendre et coordonner leur travail. Ce document n'est que rarement vu comme le code squelette auquel les développeurs doivent rajouter des informations liées à la phase d'implantation du contrôleur.
- Troisièmement, même si des outils permettent de passer d'une spécification à un squelette de code, le mécanisme inverse est notablement plus complexe à mettre en œuvre. Ainsi il est souvent impossible de retranscrire l'impact des décisions prises au niveau de la phase de programmation au niveau des modèles de spécification.

Pour pallier ce problème, les roboticiens utilisent souvent des langages comme ESTEREL [21] ou les Statecharts [69], car leur sémantique formelle permet de rendre exécutables des modèles comportementaux (via génération de code/compilation). De plus, des techniques d'analyse peuvent être utilisées sur ces modèles de manière à détecter au plus tôt les incohérences et les erreurs de modélisation. Enfin, certains de ces langages, à l'image des StateCharts [68], ont l'avantage d'être manipulables sous une forme graphique. Ces langages sont aujourd'hui utilisés dans l'industrie, mais relèvent d'une approche synchrone (où le temps est discrétisé) :

- Le temps d'exécution de toutes les actions liées à des changements d'états doit être suffisamment petit, comparé au temps couvert par l'intervalle séparant deux instants discrets, pour le considérer comme nul. Les actions sont donc condirérées comme de durée nulle et sont atomiques (ininterruptibles) car le système n'est pas sensible à de nouveaux événements entre deux instants discrets. C'est une hypothèse difficile à accepter dans notre cas, puisque les entités de contrôle s'exécutent sur des horizons à court, long ou moyen terme, et leurs actions ont, en conséquence, des temps d'exécution très variables (de quelques millisecondes à quelques secondes voire minutes pour des mécanismes de planification complexes). Si l'intervalle entre deux instants discrets est suffisamment grand pour rendre compatible l'exécution des couches hautes avec l'hypothèse synchrone, la réactivité des couches basses n'est plus assurée. Si il est suffisamment petit pour garantir la réactivité des couches basses, le temps d'exécution des actions des couches hautes viole l'hypothèse synchrone.
- D'autre part l'hypothèse synchrone n'est pas sans poser de problème dans le cadre d'une architecture logicielle distribuée (la solution est alors basée sur une approche mixte synchrone/asynchrone).

Nous nous orientons donc vers une approche asynchrone, qui permet de préserver la sensibilité "permanente" du système aux événements, selon la densité du temps. L'approche asynchrone impose de gérer la préemption potentiellement induite puisque les actions peuvent (logiquement à notre avis) être interrompues. ELECTRE [39] est un exemple de langage asynchrone, déjà utilisé conjointement avec un formalisme objet tel les objets à contrôle réactif [22] [12]. Le formalisme retenu doit être adapté au principe de raffinement du langage CoSARC et doit posséder un pouvoir d'expression adapté à la description de comportements et des interactions entre les entités de contrôle : il doit en particulier prendre en compte le parallèlisme et la concurrence. Nous avons choisi d'utiliser le formalisme des Réseaux de Petri à Objets (RdPO, OPN en anglais) [100] qui est une des nombreuses extensions des réseaux de Petri. Le choix des RdPO est guidé par plusieurs considérations, qu'aucun autre formalisme ne satisfait à notre connaissance :

1. Le formalisme des RdPO est associé à une notation graphique qui permet de représenter et de manipuler facilement les RdPO.
2. L'expressivité de la notation est adéquate à la description précise du parallèlisme, de la concurrence et de la synchronisation. Par ailleurs la notation permet, à la différence des machines à état, de représenter le flux de contrôle (le graphe de contrôle du RdPO) et le flux de données (les objets) sur le même modèle.
3. Les réseaux de Petri (plus rarement les RdPO), sont utilisés dans les domaines de l'automatique et de la robotique [40] [55]. Les experts de ces domaines sont donc habitués à leur utilisation,

et la prise en main du langage à composants ne devrait en être que plus aisée.

4. Les RdPO peuvent être vus comme un langage de programmation. Il est possible de générer, à partir de la description d'un RdPO, le code exécutable correspondant [15]. Ceci permet de limiter l'effort des programmeurs pour transposer le modèle d'un RdPO en code et permet d'éviter les erreurs humaines de traduction.
5. Le formalisme des RdPO est largement étudié et de nombreux travaux s'attachent à l'exploiter, de manière à fournir des techniques d'analyse. Ces analyses portent sur différents aspects en fonction de la classe de RdP utilisée : analyse des états logiques, vérification de scénarios temporels [116], [93]. Si nos travaux n'ont pas directement trait à l'analyse ou à la validation de systèmes temps-réel, il convient de fournir une solution qui permette, dans le futur, d'exploiter les nombreux travaux ayant déjà fait leur preuve dans ces domaines.

Nous proposons d'étendre la notation des RdPO de manière à l'intégrer avec les autres notations proposées par le langage CoSARC. Cette extension des RdPO permet de modéliser et programmer les comportements des entités de contrôle : elle permet de représenter les points d'échanges de messages et propose un mécanisme de composition, basé sur ces points d'échanges de messages, permettant de synthétiser un RdPO à partir de plusieurs RdPO (formalisation de la composition). En intégrant les RdPO au langage CoSARC, nous pensons pouvoir atténuer la principale faiblesse de cette notation : l'explosion de la taille des modèles qui les rend difficile à visualiser et à manipuler dès lors qu'ils sont utilisés pour modéliser le comportement d'une application dans sa globalité. Nous pensons que diviser le RdPO global en RdPO de taille plus modeste (chacun associé à la description d'une entité de contrôle ou d'une interaction) permet de réduire significativement la complexité d'écriture du comportement global et favorise par ailleurs la réutilisation de ses parties.

7.1.4 Séparation des préoccupations

La séparation des préoccupations est un paradigme actuellement très populaire dans le monde de la recherche, sur lequel sont basés beaucoup d'espoirs. Le langage à composants CoSARC intègre ce paradigme, mais d'une façon plus rigide (et dédiée au domaine visé - i.e. la robotique) que dans les approches classiques telle la programmation par aspects (AOP). Le langage définit un ensemble de catégories de composants (cf. fig. 7.3), chaque catégorie permettant de décrire un aspect d'un contrôleur.

Premièrement, le langage à composants permet de séparer la description des comportements *métiers* des entités de contrôle, de la description des protocoles d'interaction sur lesquels ils reposent. Les comportements *métiers* sont décrits à partir de composants spécifiques appelés *composants de contrôle* et les protocoles sont décrits à partir de composants appelés *composants connecteurs* (en référence aux ADL). La séparation entre ces deux aspects est primordiale dans le domaine d'application ciblé, car les interactions entre entités de contrôle peuvent être potentiellement très complexes, induisant des synchronisations et des échanges de messages multiples. La possibilité de décrire et de réutiliser les protocole d'interaction indépendamment des comportements *métiers* devrait permettre, à terme, un gain de productivité. D'autre part, cette séparation diminue la granularité des composants et simplifie de fait l'écriture de leurs RdPO (leur comportement indivuel étant moins complexe).

Deuxièmement, le langage à composants permet de séparer la description des entités de contrôle, de celle des entités représentant les connaissances qu'elles manipulent (dans leur comportement) et échangent (dans leurs interactions). Le langage CoSARC définit en cela une troisième catégorie de composants appelés *composants de représentation*. De ce fait, le langage tire parti des conclusions issues de l'étude des architectures de contrôle (cf. chapitre 3). Séparer ces deux aspects nous semble nécessaire car les entités représentant les connaissances manipulées doivent être indépendantes de

la manière dont une entité de contrôle utilise ces connaissances. Elles permettent de représenter, sous différentes formes, le monde sur lequel le contrôleur agit (e.g. partie opérative, environnement, capteurs/actionneurs, événements, données d'état, etc.) ou les stratégies dont il dispose pour gérer la dynamique de ce monde (générateurs de trajectoire, lois mathématiques, etc.). Une entité de contrôle décrit la façon dont sont utilisées certaines de ces connaissances pour imposer au robot un comportement global. La réutilisation des *composants de représentation* est influencée par la nature du monde sur lequel le contrôleur agit, elle est en général conditionnée par la partie opérative du robot et par le type d'environnement physique dans lequel le robot évolue. La réutilisation des *composants de contrôle* est plutôt influencée par les activités qu'un robot doit réaliser, elle se fait donc plutôt entre des robots ayant des (parties de) comportements proches. Séparer les deux aspects nous semble également nécessaire étant donné la nature des entités de représentation des connaissances et des entités de contrôle : les premières sont vues comme des entités passives (ne s'exécutant que sur demande de réalisation d'un de leur service, et de façon séquentielle) alors que les secondes sont vues comme des entités actives (s'exécutant en "continu" et potentiellement de façon parallèle).

Troisièmement, le langage à composants permet de séparer la description fonctionnelle d'une architecture logicielle, de la description du déploiement de cette architecture sur une infrastructure matérielle/logicielle, au-dessus de laquelle elle s'exécute. L'aspect "déploiement" est géré à travers la catégorie des composants appelés *configurations*. Ces composants sont chargés de décrire des architectures déployées, c'est-à-dire qui prennent en compte un ensemble de contraintes liées à l'utilisation de services techniques. Le déploiement est primordial dans le cadre des systèmes réactifs (et donc de la robotique) qui requièrent une gestion précise de l'exploitation des ressources matérielles et système disponibles sur l'infrastructure matérielle/système (e.g. ordonnanceurs, capteurs/actionneurs, ressources de communication réseau, etc.). Nous limitons pour l'instant notre réflexion à la problématique du placement des composants sur une infrastructure matérielle et à leur possibilité d'ordonnancement.

La séparation de ces différentes préoccupations à travers la manipulation de composants de natures différentes est la base du langage CoSARC. Les sections suivantes présentent en détail chacune des quatre catégories de composants manipulables dans le langage : composants de représentation, composants de contrôle, composants connecteurs et composants de configuration. Chaque catégorie est décrite par un meta-modèle de descripteurs et un meta-modèle de composants qui sont des extensions (cf. fig. 7.3) des meta-modèles présentés en début de chapitre. Des exemples de syntaxe graphique et textuelle sont donnés, respectivement pour les composants et les descripteurs.

7.2 Composants de Représentation

7.2.1 Introduction

La première catégorie de composants proposée dans le langage CoSARC est celle des *composants de représentation*. Ces composants sont utilisés pour décrire l'ensemble des connaissances qu'un contrôleur possède sur le "monde réel", c'est-à-dire sur le robot et son environnement. Ils permettent de représenter des concepts abstraits comme des événements, des données d'état (données reconstituées, données capteur), des données de commande actionneur, des cartes de l'environnement, des stratégies d'observation de l'environnement, des stratégies de commande de la partie matérielle, etc. Ils servent également à représenter des concepts concrets, comme les éléments matériels constituant la partie opérative d'un robot (.e.g. bras mécanique, capteurs, actionneurs, véhicule, etc.).

Les attributs de ces composants permettent, par exemple, de représenter les propriétés physiques, géométriques et cinématiques, des éléments matériels. Leurs ports fournis permettent d'accéder à certaines de ces propriétés ou de réaliser des calculs qui s'appuient sur ces propriétés. A titre d'illustration, un *composant de représentation* est utilisé pour représenter le véhicule du robot manipulateur

mobile. Ce composant `Véhicule` regroupe l'ensemble des spécificités physiques du véhicule considéré ainsi que les services de calcul associés (un changement de référenciel, par exemple). Un *composant de représentation* est utilisé pour représenter un événement, lié à l'approche d'un obstacle. Ce *composant de représentation* `événement Obstacle Approche` contient les données sur laquelle se base la détection de l'obstacle, ainsi que la date d'occurrence de l'événement, et fournit un service de détection de l'événement. Le composant `événement Obstacle Approche` possède un port requis, à travers lequel il utilise certains des services du `Véhicule`, pour mettre en œuvre la détection.

Les *composants de représentation* sont des entités passives, en ce sens qu'ils n'exécutent leurs activités que lorsqu'ils sont sollicités, à travers l'appel d'un des services qu'ils fournissent. Les communications établies lors des appels sont uniquement synchrones. Dans ce cas l'appelant du service est bloqué jusqu'à la terminaison des traitements réalisés par le fournisseur du service.

7.2.2 Meta-modèle

Le meta-modèle de *descripteur de composant de représentation*, décrit dans la figure 7.7, étend le meta-modèle de descripteur de composant (la notation **redefines** est celle de UML 2.0). Un *descripteur de composant de représentation* possède un ensemble de *descripteurs de port de représentation* qui référencent chacun une *interface de représentation*. Une *interface de représentation* est une interface composite qui contient un ensemble d'*interfaces synchrones* (qui correspondent à des signatures d'opérations). Un *descripteur de port de représentation* est composé d'un ensemble de *descripteurs de port synchrone* qui référencent chacun une *interface synchrone*, elle-même contenue dans l'*interface de représentation* qu'il référence.

Les propriétés internes des *descripteurs de composant de représentation* sont des attributs et des opérations. Les attributs peuvent être de type primitif (entier, flottant, etc.), d'un type défini par un *descripteur de composant de représentation* ou de type *connexion de Representation* (cf. fig. 7.9). Si un ou plusieurs des attributs a pour type, un type défini par un *descripteur de composant de représentation*, ses instances seront des composites. Les opérations peuvent amener la modification de certains de ses attributs, et donc, potentiellement, celle de l'architecture agrégée. Elles peuvent faire directement appel aux services fournis par les *ports de représentation fournis* de ses attributs. En d'autres termes, le composite est implicitement connecté à tous les ports fournis des *composants de représentation* qu'il agrège.

A partir des contraintes exprimées sur le modèle de la figure 7.7 on déduit implicitement deux catégories d'*interface de représentation* : celles associées aux *descripteurs de port fourni* ne contiennent que des *interfaces synchrones* de qualité "In" (réception de message) ; celles associées aux *descripteurs de port requis* ne contiennent que des *interfaces synchrones* de qualité "Out" (émission de message). Ainsi, chaque *interface synchrone* référencée par un *descripteur de port synchrone fourni*, est implantée par une opération définie dans le *descripteur de composant de représentation*. Un *descripteur de port synchrone requis* correspond à l'appel d'un service implanté par un autre *composant de représentation*. Une opération définie au sein d'un *descripteur de composant de représentation* peut, si besoin est, utiliser les *descripteurs de port synchrone requis* pour réaliser des appels à des services fournis par d'autres *composants de représentation*.

Les *composants de représentation* (cf. fig. 7.8) sont des instances de *descripteurs de composant de représentation*. Les ports des *composants de représentation* sont des ports composites appelés *ports de représentation*, instances de *descripteurs de port de représentation*. Un *port de représentation* est lui même composé d'un ensemble de *ports synchrones*. Il faut noter que chaque *descripteur de port de représentation* et chaque *descripteur de port synchrone* ne peut être instancié qu'une seule fois, pour chaque instance d'un *descripteur de composant de représentation*.

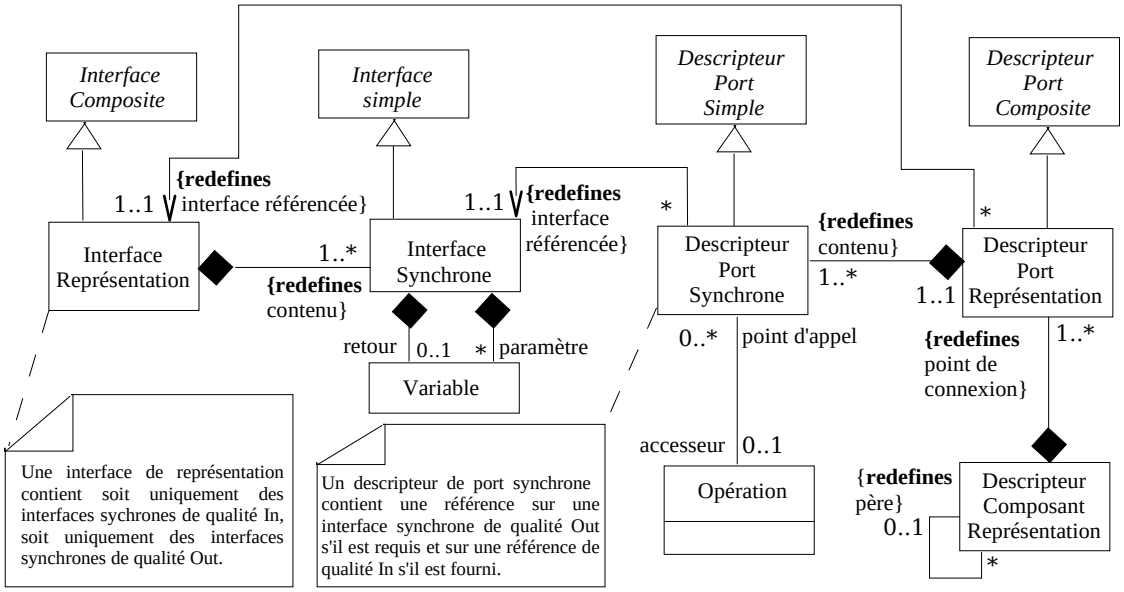


FIG. 7.7 – Meta-modèle des descripteurs de composant de représentation

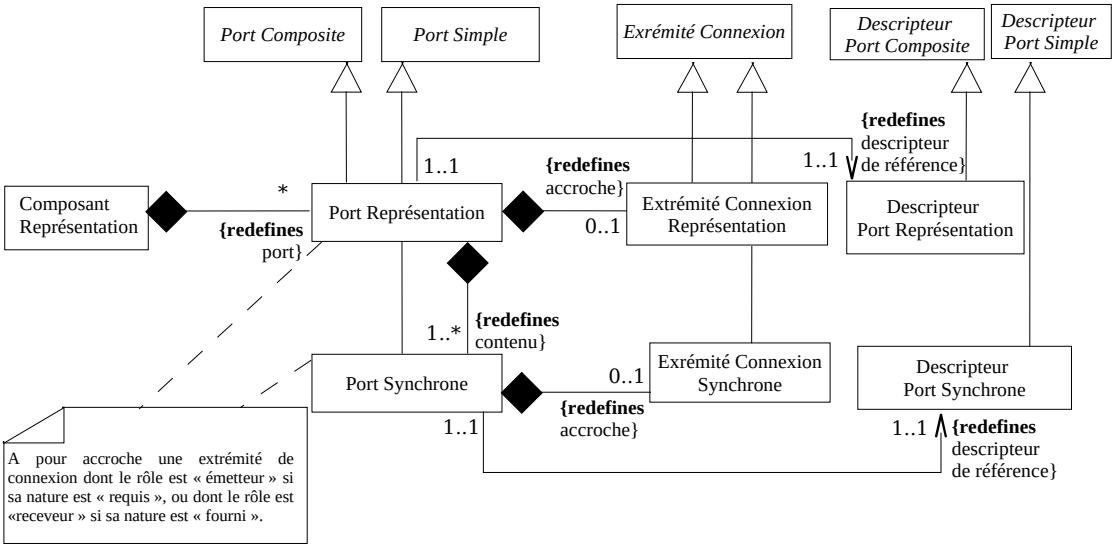


FIG. 7.8 – Meta-modèle des composants de représentation

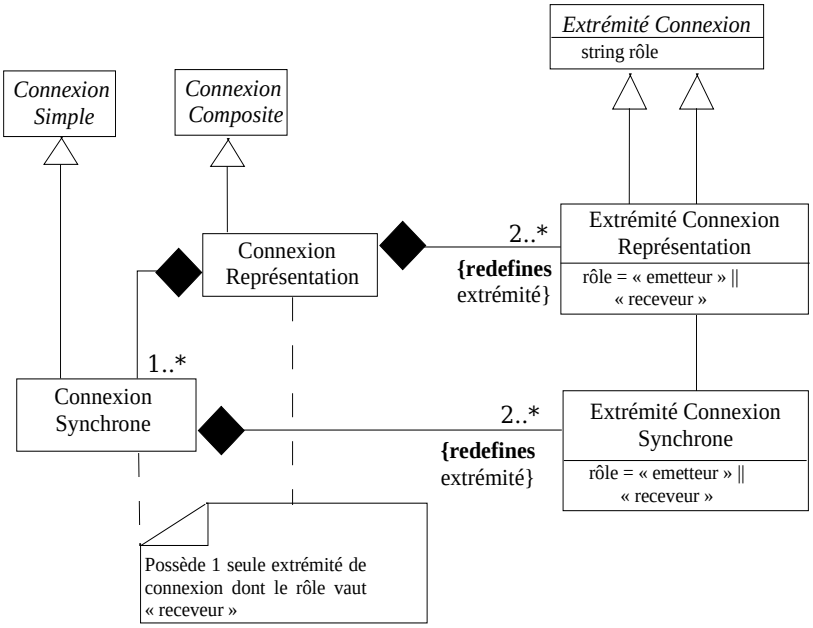


FIG. 7.9 – Meta-modèle : connexions synchrones

Les *ports de représentation* sont connectés via des connexions appelées *connexions de représentation* et les *ports synchrones* sont connectés via des connexions appelées *connexions synchrones* (cf. fig 7.9 et fig. 7.8). Une *connexion de représentation* est constituée d'un ensemble de *connexions synchrones*. A partir des contraintes exprimées sur le meta-modèle, on déduit que ces connexions permettent de connecter au moins un port requis avec un unique port fourni. Concrètement, cela exprime le fait que plusieurs *composants de représentation* peuvent utiliser les services offerts par un *composant de représentation*. Les ports requis sont connectés à des extrémités de connexion dont le rôle est *émetteur* et les ports fournis sont connectés à des extrémités de connexion dont le rôle est *receveur*. Les rôles associés aux extrémités des connexions des deux types sont donc prédéfinis.

A partir de l'ensemble des contraintes exprimées sur le meta-modèle, il est possible de "masquer" la complexité des connexions entre *composants de représentation*. En effet, lorsqu'une *connexion de représentation* est établie, l'ensemble des *connexions synchrones* qui la constitue est automatiquement déduit des ports synchrones fournis et des ports synchrones requis, contenus respectivement dans le *port de représentation fourni* et les *ports de représentation requis* connectés.

Pour conclure, nous constatons que les *composants de représentation* ont un modèle de composition proche de celui des objets (à l'indirection induite par l'existence explicite des ports et connexions près). Ce choix a été fait par souci de simplicité, puisque nous partons de l'hypothèse que les interactions entre *composants de représentation* seront simples (essentiellement des demandes de calcul ou d'accès à certains attributs).

7.2.3 Exemple de syntaxes

Un exemple de *descripteur de composant de représentation* est donné, sous forme de syntaxe textuelle et un exemple de *composant de représentation* instance de ce descripteur, est donné sous forme

de syntaxe graphique. Le descripteur `CompExemple` (cf. fig. 7.10) présente les éléments syntaxiques essentiels du langage. Il contient trois parties : la définition des descripteurs de ports, des attributs et des opérations. La figure 7.11 présente un *composant de représentation* instance de `CompExemple`.

Dans la partie de description des ports, `CompExemple` contient quatre *descripteurs de port de représentation*, notés `P1`, `P2`, `P3` et `P4`. Le mot clef utilisé pour les déclarer est `representation port`. Chaque *descripteur de port de représentation* référence une *interface de représentation*. La référence à une interface est déclarée via le mot clef `refers`. Par exemple, `P1` fait référence à l'*interface de représentation* `I1`. Chacun de ces *descripteurs de port de représentation* est décomposé en un ou plusieurs *descripteurs de port synchrone* (déclarés via le mot clef `synchronous port`). Chaque *descripteur de port synchone* fait référence à une *interface synchone* (mot-clef `refers`). Par exemple, le *descripteur de port synchrone* `ps1` fait référence à l'*interface synchrone* `void service1()`, elle-même contenue dans l'interface `I1` (cf. fig. 7.11). Les *descripteurs de port de représentation* sont qualifiés de fournis ou requis, respectivement via l'utilisation des mot-clefs `provided` ou `required`. Les *descripteurs de port synchrone* sont (automatiquement) qualifiés de la même façon que les *descripteurs de port de représentation* qui les contiennent : `ps1` est un descripteur de port fourni, comme `P1` qui le contient. Lorsqu'un *descripteur de port de représentation* est fourni, chacun des *descripteurs de port synchrone* qu'il contient doit être associé à une opération implantant l'interface synchrone qu'il référence. Par exemple, le *descripteur de port synchrone* `ps1` est associé à l'opération `void do1()` (définie dans la partie `Opération` de `CompExemple`) qui implante l'interface `I1.service1()`. Une même opération peut-être utilisée pour implanter différentes *interfaces synchrones*, par exemple `void do1()` sert aussi à implanter `I2.service1()`. Lorsqu'un *descripteur de port de représentation* est requis, il est utilisé au sein de certaines opérations définies dans le descripteur, pour réaliser des appels de services. Pour cela, le code des opérations fait directement référence au *port synchrone* requis. Par exemple, le corps de l'opération `do1()` accède au service `I3.service3()` via le port `ps4`. Certaines opérations peuvent n'être rattachées à aucun *descripteur de port de représentation* fourni, elles permettent de réaliser des calculs intermédiaires, internes à un composant.

Les *interfaces de représentation* (présentées dans la figure 7.11) ne sont pas définies à l'intérieur du *descripteur de composant de représentation*. C'est pourquoi ces interfaces ont besoin d'être importées avant la définition des différents ports du composant. Ceci est effectué via l'utilisation du mot-clef `import interface`. De la même façon, dans la partie `Attributes` (où sont définis les attributs), il est nécessaire d'importer les définitions des descripteurs de composants utilisés pour créer les *composants de représentation* attributs de `CompExemple`. Ceci est réalisé via le mot-clef `import descriptor`. Cette partie `Attributes` peut donc contenir des attributs de types *composant de représentation* (`attr2` et `attr3` sur l'exemple), des attributs de type primitif (`attr1`) et des attributs de type *connexion de représentation* (`attr4`). Les attributs de type *connexion de représentation* permettent de représenter et de manipuler les relations entre *composants de représentation*, ce qui permet de gérer (recherche, création, destruction) de façon dynamique leur assemblage. Ceci laisse également la possibilité de reconstituer, à tout moment de l'exécution, le graphe de composition contenu dans un composite.

Dans la partie `Opérations` d'un descripteur, deux opérations sont obligatoirement présentes : l'opération d'initialisation (notée `init()`), appelée lors de la création d'une instance d'un *descripteur de composant de rerésentation*, et l'opération de fin (notée `end()`) appelée à sa destruction. L'opération d'initialisation de `CompExemple` instancie des *composants de représentation* (mot-clef `create`) et les assemble (mot-clef `connect`) par une *connexion synchrone de représentation* (`RepresentationConnection`) entre leurs *ports de représentation* (`portY` et `portX`). Lorsqu'une *connexion de représentation* est créée, elle est composée d'un ensemble de *connexions synchrones*. Ces *connexions synchrones* ne sont pas décrites au niveau de la syntaxe textuelle, car elles sont automatiquement déduites de la connexion et de la structure des *ports de représentation* qui les contiennent. On remarque qu'à chaque connexion, les rôles associés aux extrémités connectées aux ports sont spécifiées (mot-clef `as` suivi du nom du rôle). L'opération de fin détruit la composition

```
Representation Component Descriptor CompExemple
{
ports:
 import interface I1, I2, I3, I4;
 provided representation port P1 refers {I1}{
  synchronous port ps1 refers {In void service1()} : void do1();
 }
 provided representation port P2 refers {I2}{
  synchronous port ps2 refers {In void service1()} : void do1();
  synchronous port ps3 refers {In int service2()} : int getattr1();
 }
 required representation port P3 refers {I3}{
  synchronous port ps4 refers {Out void service3()};
 }
 required representation port P4 refers {I4}{
  synchronous port ps5 refers {Out int service4()};
 }

Attributes:
import descriptor CR1, CR2;
int attr1; CR1 attr2; CR2 attr3;
RepresentationConnection attr4;

Operations:

init() //initialisation
{
 attr1 = 0;
 attr2 = create CR1();  attr3 = create CR2();
 //connexion des ports des composants de représentation contenus
 attr4 = connect attr2.portY as emitter, attr3.portX as receiver;
 //appel direct d'un service d'un composant de représentation contenu
 attr2.portX.port1();
}

end() //destructeur
{
 disconnect attr4;
 destroy attr2();
 destroy attr3();
}

void do1( )
{
 //code de do1
 ps4();//appel d'un service via le port ps4
 //..
 int temp = ps5();//appel de service via le port ps5
 //...
}

int getAttr1()
{
 return attr1;
}
};
```

FIG. 7.10 – Exemple d'un descripteur de composant de représentation

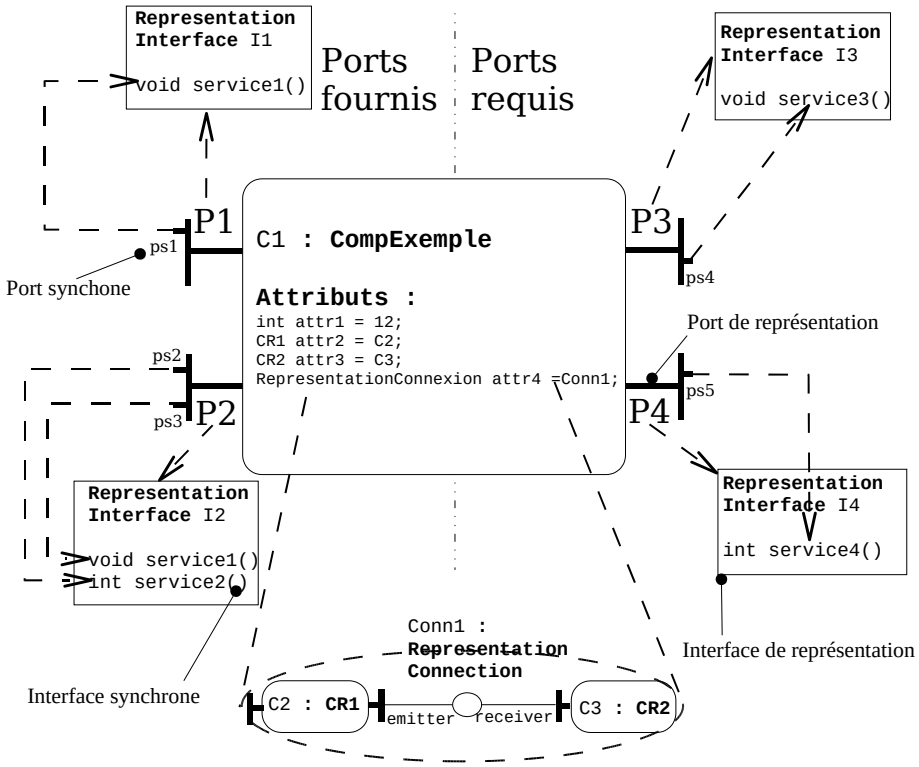


FIG. 7.11 – Exemple de composant de représentation instance

ainsi créée, en détruisant les *composants de représentation* agrégés dans le composant `CompExemple` (mot-clef `destroy`), ainsi que leur connexion (mot-clef `disconnect`). Il n'y a qu'une seule opération de fin, mais plusieurs opérations d'initialisation peuvent être définies.

7.3 Composants de Contrôle

7.3.1 Introduction

La deuxième catégorie de composants proposée par le langage COSARC est celle des *composants de contrôle*. Ils sont utilisés pour décrire les entités de contrôle à partir desquelles est décrit le comportement d'un contrôleur. Les *composants de contrôle* sont utilisés, dans le modèle d'architecture de CoSARC, pour représenter les entités de contrôle simples (cf. chapitre 6) : les superviseurs globaux, les superviseurs de ressource, les modes, les actions, les commandes, les perceptions, les générateurs d'événements, ainsi que les contrôleur d'entrées/sorties. Leurs activités consistent à percevoir l'état de l'environnement ou de sa partie opérative (via des données d'état ou des événements), à prendre des décisions en fonction de ces perceptions et de leur état courant, et à utiliser les services fournis par des composants de contrôle (conceptuellement situés dans des couches plus basses) afin de réaliser leurs intentions.

Les *composants de contrôle* sont des entités actives, dans le sens où ils possèdent chacun une activité propre tout au long de leur cycle de vie, sans que cette activité soit soumise à l'appel d'un des services qu'il fournit. L'activité des *composants de contrôle* est asynchrone et parallèle. Les communications entre ces composants sont asynchrones, ce qui implique que leur activité n'est pas bloquée lorsqu'ils interagissent. Le parallélisme est dû au fait qu'un *composant de contrôle* peut, à un moment donné de son exécution, réaliser différents calculs en même temps. Le comportement asynchrone d'un *composant de contrôle* exprime la façon dont sont organisées ses activités internes, il exprime l'accès concurrent aux données et la synchronisation de ses activités parallèles internes. Un *comportement asynchrone* décrit également la façon dont le *composant de contrôle* synchronise ses activités en fonction des messages qu'il reçoit et émet à travers ses ports. Il exprime les différents chemins d'exécution possibles pour réaliser un service, c'est-à-dire les différentes séquences de déclenchement de ses activités internes.

La notation retenue pour décrire le comportement asynchrone d'un composant de contrôle est celle des réseaux de Petri à Objets (RdPO), que nous enrichissons afin de la rendre compatible avec les autres notations du langage CoSARC. Cette notation a plusieurs caractéristiques qui nous semblent adaptées à la modélisation des *comportements asynchrones* des entités de contrôle :

- Elle permet de représenter le flux de contrôle (la structure du RdPO) et le flux de données (les objets contenus dans les jetons qui transitent dans les RdPO) sur un même modèle, en exprimant précisément les synchronisations d'activités parallèles et la concurrence d'accès aux données.
- Elle permet d'exprimer les activités réalisées par un *composant de contrôle* en fonction de son état abstrait qui est facilement lisible sur le modèle. L'état abstrait est définit par le vecteur de marquage du RdPO (i.e. la distribution des jetons dans les places) et par l'état des objets contenus dans ces jetons.
- Elle permet de définir des fenêtres temporelles, qui décrivent implicitement la génération d'événements temporels. Ces événements temporels servent à déclencher des cycles d'asservissement (permettant ainsi d'obtenir un comportement périodique) ou à exprimer des "chiens de garde" (surveillance de la durée d'activités).

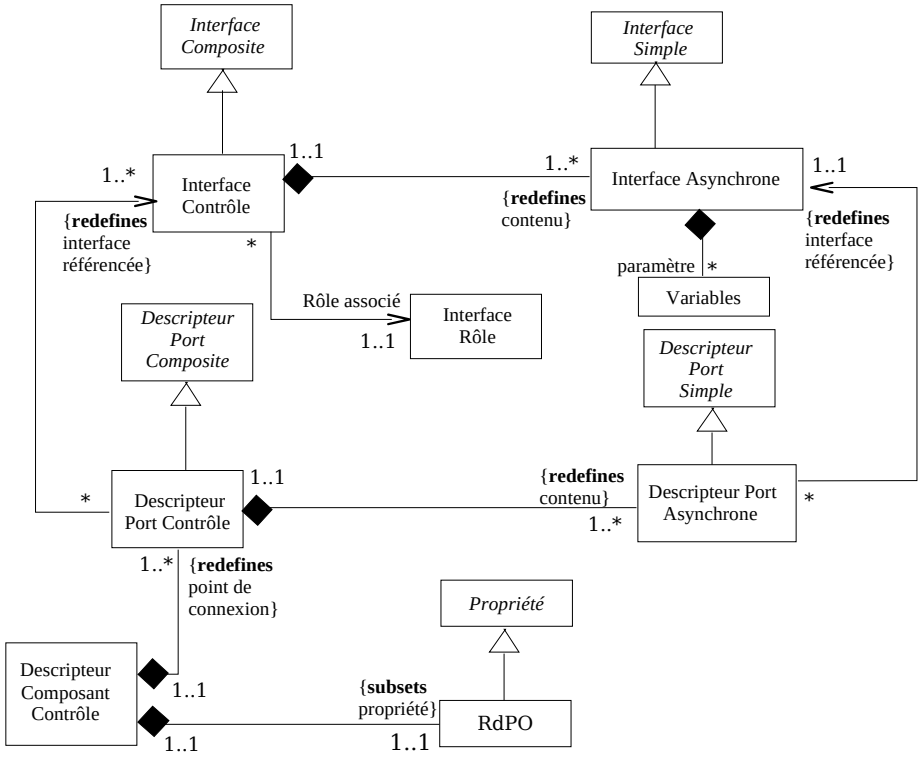


FIG. 7.12 – Meta-modèle des descripteurs de composants de contrôle

7.3.2 Meta-modèle

Un *descripteur de composant de contrôle* (cf. fig. 7.12) contient un ensemble de *descripteurs de port de contrôle* qui référencent chacun une *interface de contrôle*. Un *descripteur de port de contrôle* est composé d'un ensemble de *descripteurs de port asynchrone* qui référencent chacun une *interface asynchrone* contenue dans l'*interface de contrôle* qu'il référence. Chacun de ces descripteurs de ports donne naissance à un unique port, pour chaque *composant de contrôle* instance.

Les *interfaces asynchrones* sont des interfaces simples qui correspondent soit à des services d'émission (`Out`) ou de réception (`In`) asynchrone de messages (appels de services non bloquants pour l'émetteur). Une *interface de contrôle* représente l'utilisation ou l'offre d'un service, réalisée autour d'échanges de messages multidirectionnels (aucune contrainte sur la *qualité* des *interfaces asynchrones* contenues). A chaque *interface de contrôle* est associée une unique *interface de rôle*. Une *interface de rôle* définit le rôle que joue un *composant de contrôle* dans le cadre d'une connexion entre *ports de contrôle*. Les *interfaces de rôle* et le mécanisme de connexion sont présentés dans la section suivante.

Les propriétés internes des *descripteurs de composant de contrôle* sont des attributs, des opérations et un *comportement asynchrone* décrit par un RdPO. Les attributs sont soit de type primitif, soit d'un type défini par un *descripteur de composant de représentation*, soit de type *connexion de représentation*. Ainsi, un *composant de contrôle* est nécessairement un composite, qui agrège un ensemble de *composants de représentation*, mais il faut noter qu'un *composant de contrôle* ne peut pas agréger des *composants de contrôle* (puisqu'ils ne peuvent contenir d'attributs dont le type est défini par un *descripteur de composant de contrôle*). La notation RdPO utilisée est décrite dans la figure

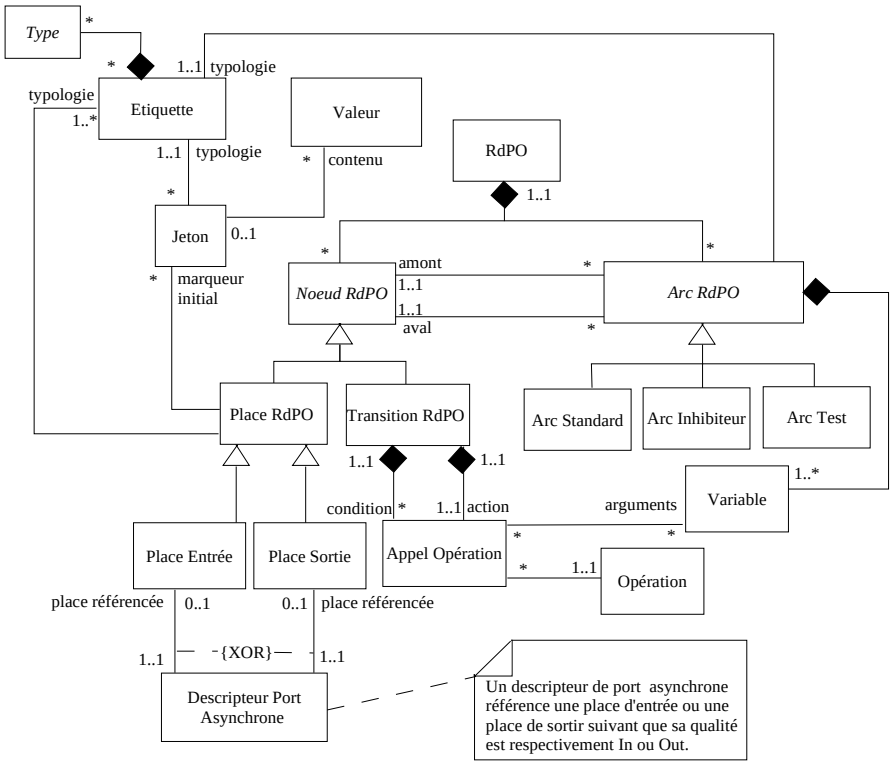


FIG. 7.13 – Meta-modèle : réseaux de Petri à Objets

7.13. Un RdPO possède un graphe de contrôle dont les noeuds sont des places et des transitions, les arcs permettant de relier les places aux transitions. Les transitions d'un RdPO réalisent des appels aux opérations définies au sein du *descripteur de composant de contrôle* pour lequel ce RdPO a été décrit. Les places contiennent des jetons, qui eux-mêmes contiennent des *composants de représentation*. Il existe des places spécifiques appelées *places d'entrée* (en grisé) et *places de sortie* (en noir) (cf. fig. 7.16). Ces places permettent de décrire respectivement des points de réception et des points d'envoi de messages. Chacune d'entre elles est reliée à un unique *descripteur de port asynchrone*, qu'il soit fourni ou requis. Lorsqu'une *place d'entrée* ou une *place de sortie* contient un jeton, cela représente le fait que le message correspondant à la place est respectivement reçu ou émis. Les données associées aux messages sont les jetons, qui sont des tuples de *composants de représentation*. Un *descripteur de port de contrôle* est indirectement relié à un ensemble de places d'entrée et de sortie, qui correspondent à celles reliées aux *descripteurs de port asynchrone* qu'il contient.

Le meta-modèle des *composants de contrôle* est présenté dans la figure 7.14. Les *ports de contrôle* sont des ports composites, instances des *descripteurs de port de contrôle*. Un *port de contrôle* est composé d'un ensemble de *ports asynchrones* qui sont des *ports simples*. Chaque *composant de contrôle*, possède le RdPO défini au niveau de son descripteur, valué par un vecteur de marquage propre. L'opération d'initialisation permet de fixer le vecteur de marquage initial du RdPO, au moment de la création d'un *composant de contrôle* : c'est à ce moment que sont créés les *composants de représentation* qu'un *composant de contrôle* agrège. Les *composants de représentation* agrégés sont identifiés à travers des variables définies sur les arcs des RdPO, et sont manipulés au sein des transitions (appels des opérations du *descripteur de composant de contrôle*). D'autre part, les *composants de représentation* ne sont pas uniquement "locaux" à des *composants de contrôle* puisqu'ils peuvent être échangés à travers les messages émis par ces derniers.

Les connexions établies entre les *ports de contrôle* sont appelées *connexions de contrôle* (cf. fig. 7.15). Une *connexion de contrôle* permet de connecter un ou plusieurs *ports de contrôle* fournis avec un ou plusieurs *ports de contrôle* requis. Une *connexion de contrôle* est constituée d'un *composant connecteur* et d'un ensemble de *connexions d'accès de protocole*. L'assemblage des *composants de contrôle* est expliqué en détail dans la section décrivant les *composants connecteurs*.

7.3.3 Exemple de syntaxes

L'instanciation du meta-modèle permet de créer des *composants de contrôle* et leurs descripteurs. Un exemple de *descripteur de composant de contrôle* est donné, sous forme de syntaxe textuelle (cf. fig 7.16) et un exemple de *composant de contrôle* instance de ce descripteur, est donné sous forme de syntaxe graphique (cf. fig 7.17).

Le descripteur `CompControlExemple` de la figure 7.16 permet de présenter la structure typique d'un *descripteur de composant de contrôle*. Il contient quatres parties : la description des ports, la définition des attributs, l'implantation des opérations et la description d'un comportement asynchrone.

Dans la partie de description des ports, `CompControlExemple` contient deux *descripteurs de port de contrôle*, notés `P1` et `P2` (le mot-clef utilisé pour les déclarer est `control port`). Chaque *descripteur de port de contrôle* fait référence à une *interface de contrôle* (mot clef `refers`), par exemple, `P1` fait référence à l'interface `IC1` (cf. fig 7.16). Chaque *descripteur de port de contrôle* est composé d'un ou plusieurs *descripteurs de port asynchrone* (déclarés via le mot clef `asynchronous port`). Par exemple, `P1` contient `PAs1`. Chaque *descripteur de port asynchrone* référence une *interface asynchrone* qui est contenue dans l'*interface de contrôle* référencée par le *descripteur de port de contrôle* qui le contient. Par exemple, `PAs1` référence `DoIt(CR4)`, qui est contenue dans l'interface `IC1`. Les *descripteurs de port de contrôle* sont qualifiés de fournis ou requis, respectivement via l'utilisation des mot-clefs

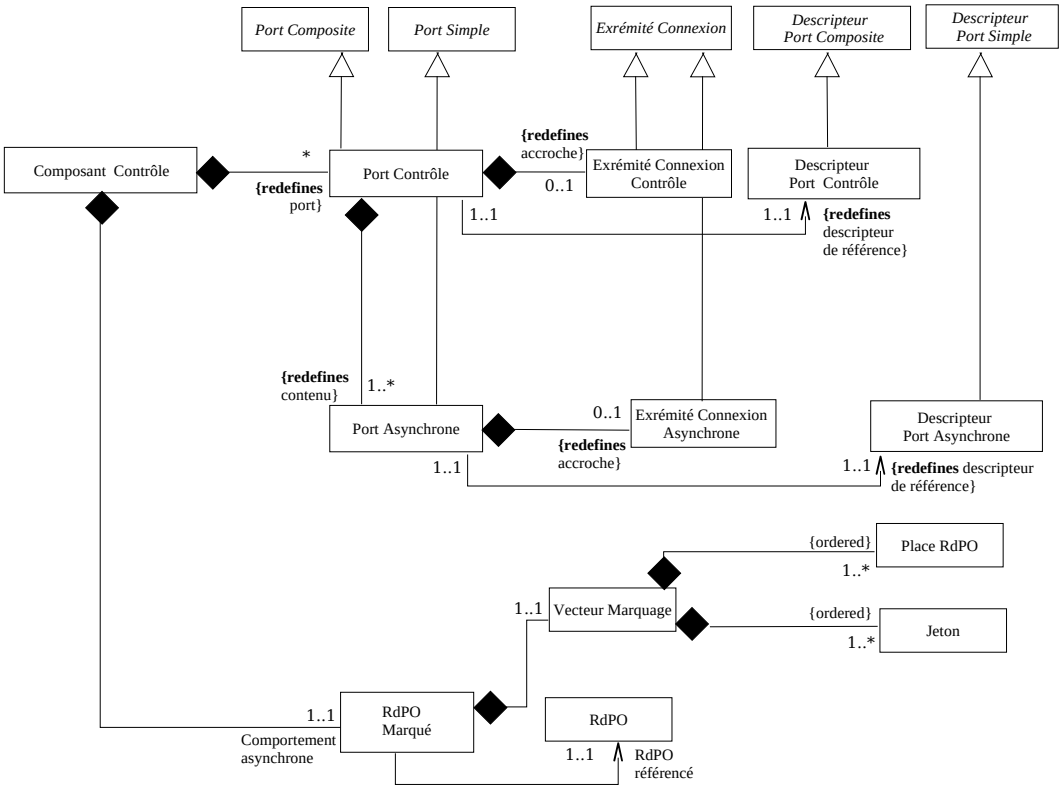


FIG. 7.14 – Meta-modèle des composants de contrôle

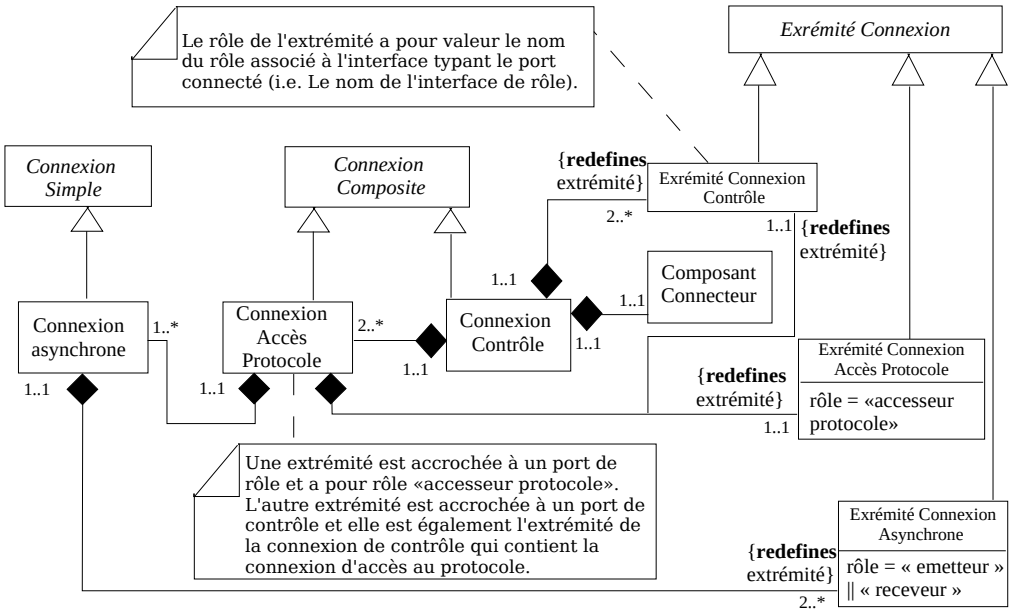
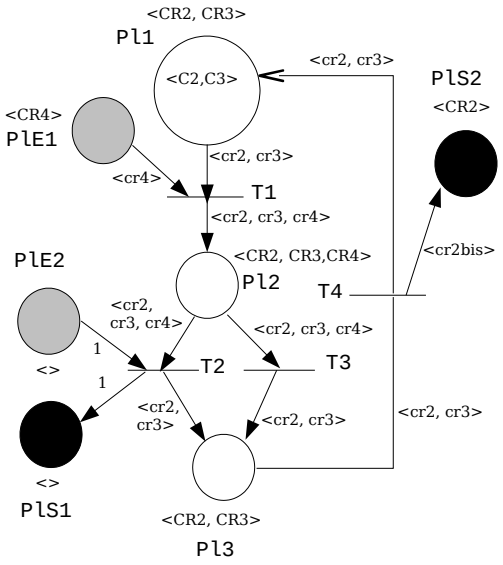


FIG. 7.15 – Meta-modèle des connexions asynchrones

```
Control Component Descriptor CompControlExemple
{
ports:
import interface IC1, IC2, IC3;

 provided control port P1 refers {IC1}{
    asynchronous port PAs1 refers {In DoIt(CR4 c4)} : PlE1;}

 provided control port P3 refers {IC3}{
    asynchronous port PAs2 refers {In DoItAgain()} : PlE2;
    asynchronous port PAs3 refers {Out AgainResponse()}: PlS1;
  }

 required control port P2 refers {IC2}{
    asynchronous port PAs4 refers {Out DoSomething(CR2 c2)} : PlS2;
  }

Attributes:
import descriptor CR1, CR2, CR3;
CR1 attrA; CR2 attrB ; CR3 attrC;
RepresentationConnection conn1;

Operations:
init(CR2 c2, CR3 c3)//constructeur
{
 //initilaisation des attributs
 attrA = create CR1();
 attrB = c2 ; attrC = c3;
 //initialisation de la connexion ...
 //initialisation du marquage initial du PNO
 addToken Pl1 <attrA,attrB>;//ajout du jeton <attrA,attrB> au marquage initial de la place Pl1
 //...
}
end()//destructeur
{
 disconnect Conn1;
 destroy attrA(); destroy attrB(); destroy attrC();
 removeTokens;
}
void do1(){...}
void do2(CR2 c2, CR4 c4){...//appels des services de composants de représentation contenus}
int computetime(){return 12;}
bool test(CR2 c) {...}
bool test2(CR3 c) {...}

Asynchronous Behaviour :
```




```
T1 :
action : do2(cr2,cr4);

T2:
action : {destroy cr4;}

T3:
time : [computetime(), inf]
conditions: {test(cr2); & test2(cr3);}
action : {do1();destroy cr4;}

T4:
action: {CR2 cr2bis = create CR2(cr2);}

};
```

FIG. 7.16 – Exemple de définition d'un descripteur de composant de contrôle

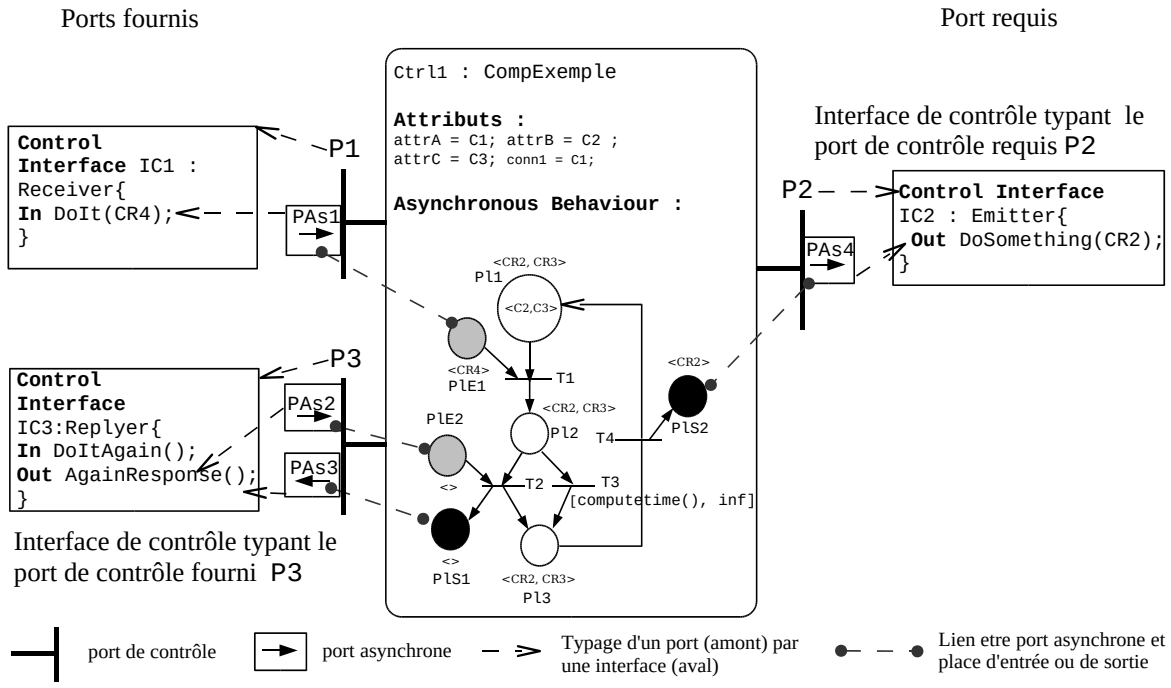


FIG. 7.17 – Exemple d'un composant de contrôle

`provided` ou `required`. Les *descripteurs de port asynchrone* sont qualifiés de la même façon que les *descripteurs de port de contrôle* qui les contiennent. Ainsi, `PAs1` est un port fourni, comme `P1`, qui le contient.

Les *interfaces de contrôle* et les *interfaces asynchrones* qu'elles contiennent (présentées dans la figure 7.17) ne sont pas définies à l'intérieur du *descripteur de composant de contrôle*, c'est pourquoi ces interfaces ont besoin d'être importées (utilisation du mot-clef `import interface`). La syntaxe pour l'import (des interfaces ou des descripteurs) est identique à celle utilisée pour les composants de représentation. Il en va de même pour la définition des attributs et des opérations (import des descripteurs).

Les figures 7.16 et 7.17 donnent un exemple de comportement asynchrone décrit par un RdPO. Chaque transition est associée à un ensemble d'appels d'opérations définies au niveau du descripteur. Ces appels sont de trois catégories : les *actions*, les *conditions*, et les *fenêtres de temps*. Les *conditions* (opérations retournant un booléen) permettent de définir une garde pour le tir d'une transition : si toutes les conditions sont respectées (les appels aux opérations retournent tous "vrai"), le tir peut avoir lieu. La partie *conditions* d'une transition peut contenir plusieurs appels à des opérations de test (e.g. transition `T3`). Les *actions* permettent d'effectuer des calculs à partir des composants de représentation manipulés, et servent également à créer (e.g. transition `T4`) ou à détruire (e.g. `transition T2`) des *composants de représentation*, ou des connexions entre ces composants. Les *fenêtres temporelles* peuvent être définies dynamiquement via l'appel d'opérations (e.g. appel de l'opération `computetime()` pour définir la borne minimale de la fenêtre, sur la transition `T3`). Cela signifie que le tir de la transition ne pourra avoir lieu avant une période calculée par `computetime()`, une fois la transition `T3` sensibilisée.

Au niveau de la définition des opérations d'initialisation et de destruction, deux mot-clefs supplémentaires à ceux disponibles pour les *composants de représentation* sont disponibles. Le mot-clef

`addToken` permet, à l'initialisation, de créer un jeton contenant des *composants de représentation* et d'ajouter ce jeton au marquage d'une place. `removeTokens` permet, à la destruction, de détruire tous les jetons du vecteur de marquage du RdPO.

7.4 Connecteurs

7.4.1 Introduction

La troisième catégorie de composants proposée par le langage COSARC est celle des *composants connecteurs*, plus simplement appelés *connecteurs*. Les *connecteurs* sont des composants utilisés dans les *connexions de contrôle*. Chaque *connecteur* établit un protocole d'interaction spécifique, utilisé par les *composants de contrôle* connectés pour communiquer et se synchroniser. L'existence de *connecteurs* facilite la réutilisation des protocoles d'interaction, indépendamment des *composants de contrôle*.

L'idée de réifier les protocoles d'interaction sous forme d'entités de première classe [98], existe déjà depuis plusieurs années dans le monde des langages de description d'architectures. Des ADL comme Wright [6], SOFA [14] ou ArchJava [5] proposent des abstractions permettant de représenter les interactions entre composants. Des variations de modèles de composants existants, comme les composants médiums [23] ou les composants d'interaction [95] (entre composants Darwin), proposent des solutions techniques pour permettre la description et la réutilisation des protocoles. Enfin, l'idée analogue consistant à décrire indépendamment les interactions et les composants, a déjà été développée dans le monde de la modélisation par objets [37] [115]. La tendance actuelle dans les approches à composants est de faire des *connecteurs* des entités exécutables à part entière [5] [46] [75], au même titre que les composants. Le connecteur prend différentes formes en fonction des approches : schéma de conception objets, métaprogrammes, filtres de composition, etc. Un panel exhaustif des différentes solutions apportées peut être trouvé dans [20]. La nécessité de réifier les protocoles comme des entités spécifiques, c'est-à-dire séparer explicitement dans le langage l'aspect "métier" de l'aspect "interaction"[106], se justifie de différentes façons dans ces approches. Nos motivations ont été les suivantes :

- Améliorer la réutilisabilité. Si les *composants de contrôle* et les *connecteurs* étaient amalgamés, il serait impossible de réutiliser indépendamment les parties des comportements des *composants de contrôle* mettant en œuvre un protocole d'interaction.
- Faciliter la définition des *comportements asynchrones*. Les parties du comportement asynchrone d'un *composant de contrôle* concernant ses interactions sont agrégées dans des *connecteurs*, si bien que les développeurs n'ont plus qu'à écrire le comportement "métier" en se basant sur les *connecteurs* disponibles. Les RdPO des comportements asynchrones seront ainsi moins complexes à écrire.
- Améliorer l'abstraction. En donnant une identité propre aux *connecteurs*, il devient plus facile de comprendre et de manipuler les interactions. En outre, obliger les développeurs à créer des *connecteurs* pour pouvoir assembler les *composants de contrôle*, les contraint à abstraire les types d'interactions qui existent dans leurs architectures, ce qui ne peut que permettre un gain de qualité de la modélisation.
- Apporter de la flexibilité à l'assemblage. Pour un assemblage de *composants de contrôle* donné, il est souhaitable de pouvoir changer le *connecteur* utilisé, par exemple si un nouveau *connecteur* jugé plus performant est disponible. Il est également souhaitable de pouvoir adapter le protocole utilisé dans un assemblage pour faire interagir des *composants de contrôle* qui n'ont pas été initialement conçus pour cela.
- Fournir un modèle d'exécution propre. Les *connecteurs* sont des entités logicielles réparties par essence. La traduction des *connecteurs* sous forme exécutable doit être contrôlable de façon à les configurer (adapter/optimiser) en fonction de leur déploiement, c'est-à-dire en fonction du déploiement des *composants de contrôle* qu'ils permettent d'assembler.

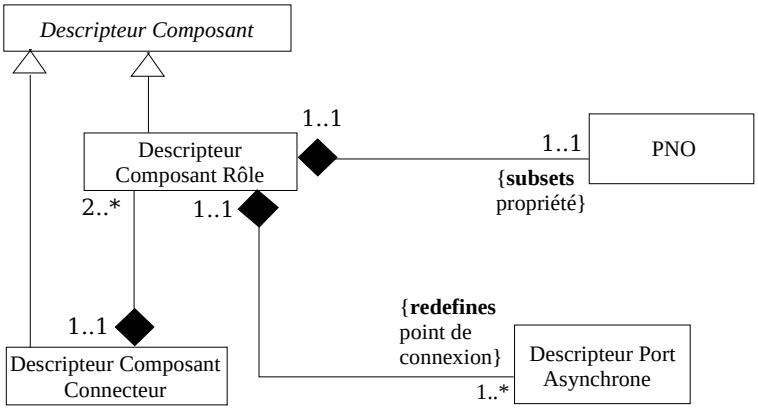


FIG. 7.18 – Meta-modèle d'un descripteur de rôle

Un des concepts centraux du concept de connexion est celui de *rôle*. Un *rôle* définit la manière dont un composant interagit dans le cadre d'une connexion donnée : on dit alors qu'un composant "joue" un *rôle*. Jusqu'à présent les *rôles* étaient fixés pour chaque type de connexion. Pour les *connexions de contrôle* nous souhaitons que les *rôles* puissent être définis par les développeurs. Pour cela, les *rôles* que peuvent jouer les *composants de contrôle* sont définis au niveau des *composants connecteurs* utilisés dans les *connexions de contrôle*. Chaque *connecteur* est composé d'un ensemble de *composants rôles* assemblés de manière à définir un protocole d'interaction. Un *composant rôle* est un composant représentant un comportement qu'adopte un *composant de contrôle* dans le cadre d'un protocole d'interaction donné. Par définition, un *connecteur* est donc un composite. Les réseaux de Petri à Objets (RdPO) permettent de modéliser le comportement asynchrone associé à chaque *composant rôle*, ce qui permet d'avoir une notation unifiée. Le RdPO d'un *connecteur* est issu de la composition des comportements asynchrones contenus dans ses *composants rôles*.

7.4.2 Meta-modèle

Un *descripteur de composant connecteur* contient un ensemble de *descripteurs de composant rôle* (cf. fig. 7.18). Il faut noter que les *descripteurs de composant rôle* sont définis à l'intérieur d'un *descripteur de composant connecteur* et ne sont donc pas considérés comme indépendants. Un *descripteur de composant rôle* contient des opérations et des attributs ainsi que la description d'un comportement asynchrone décrit par un réseau de Petri à Objets. Ce comportement asynchrone correspond à celui qu'adopte un *composant de contrôle* lorsqu'il joue un *rôle* dans une *connexion de contrôle* donnée. Un *descripteur de composant rôle* contient un ensemble de *descripteurs de port asynchrone* qui sont identiques à ceux des *descripteurs de composant de contrôle*.

Un *descripteur de composant connecteur* contient un ensemble de *descripteurs de port de rôle*. Un *descripteur de port de rôle* possède une cardinalité qui définit les contraintes quant au nombre de fois minimum et maximum respectivement où ce port doit et peut être instancié. Il référence une interface appelée *interface de rôle*, qui décrit un service d'accès au protocole définit par le *descripteur de composant connecteur*. Une telle interface représente un *rôle* que peut jouer un *composant de contrôle*. Elle contient un ensemble d'*interfaces asynchrones* qui décrivent les échanges de messages asynchrones entre le *composant connecteur* instance et des *composants de contrôle* jouant le *rôle* défini dans l'*interface de rôle*. Un *descripteur de port de rôle* contient un ensemble de *descripteurs de*

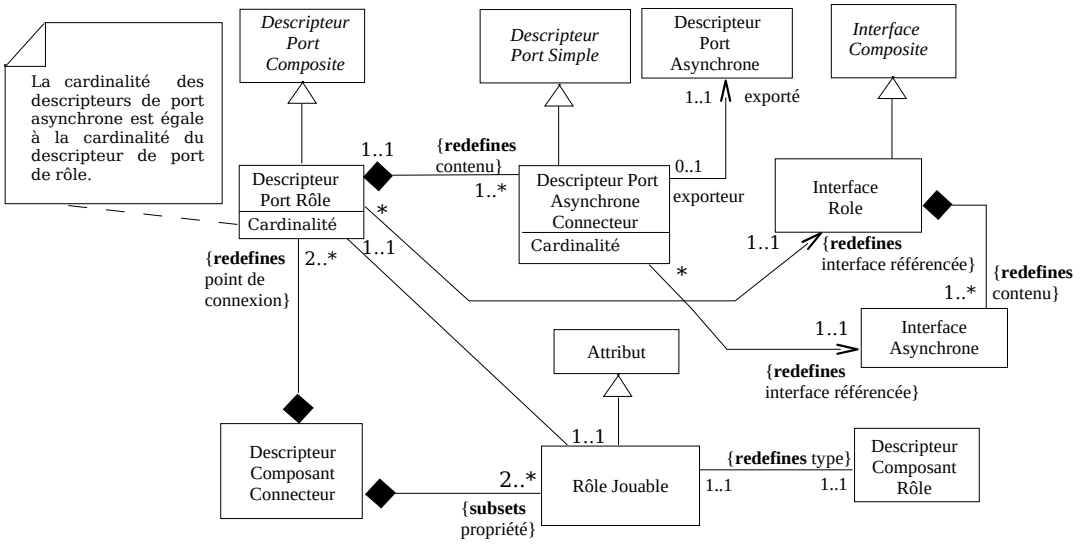


FIG. 7.19 – Meta-modèle d'un descripteur de composant connecteur

port asynchrone de connecteur qui référencent chacun une *interface asynchrone*. Un *descripteur de port asynchrone de connecteur* possède également une cardinalité (équivalente à celle du *descripteur de port de rôle* qui le contient) qui définit le nombre de fois maximal et minimal où il est instancié. Il est relié à un des *ports asynchrones* d'un des *rôles jouables* du *connecteur*.

Un *descripteur de composant connecteur* contient la définition d'un ensemble d'attributs, dont en particulier des attributs spécifiques appelés *rôles jouables*. Ces *rôles jouables* sont des attributs dont le type correspond à celui défini par un *descripteur de composant rôle* et chaque *descripteur de composant rôle* est associé à un unique *rôle jouable*. Un *descripteur de composant connecteur* définit également d'autres attributs, dont en particulier des connexions. Un *descripteur de composant connecteur* contient également (uniquement) la définition des opérations d'initialisation et de fin.

Un *composant connecteur* (cf. fig. 7.21) est instance d'un *descripteur de composant connecteur*. Il agrège un ensemble de *composants rôles* et de *connexions asynchrones simples* (cf. fig. 7.15) entre ces *composants rôles*. Un *composant rôle* (cf. fig. 7.20) possède un *port asynchrone* pour chaque *descripteur de port asynchrone* contenu dans son descripteur, ainsi qu'un comportement asynchrone (décrit par le RdPO de son descripteur) valué par un vecteur de marquage. Chaque *port asynchrone* est typé par une *interface asynchrone*. Un *composant rôle* possède également des attributs valués, ainsi que des opérations de la même manière que les *composants de contrôle*.

Un *composant connecteur* possède des ports appelés *ports de rôle*. Un *port de rôle* est typé par une *interface de rôle* et il est composé de *ports asynchrones de connecteur*. Un *descripteur de port de rôle* peut être instancié plusieurs fois de manière à donner naissance à plusieurs *ports de rôles*. Le nombre de *ports de rôle* instances d'un même *descripteur de port de rôle* correspond au nombre de fois qu'un *composant de contrôle* joue le *rôle* correspondant. Il est contraint par la cardinalité de ce descripteur. Un *rôle* est joué chaque fois qu'un *port de contrôle* (d'un *composant de contrôle* connecté via le *connecteur*) est typé par une *interface de contrôle* dont le *rôle associé* correspond à une *interface de rôle* définie dans un des *descripteurs de ports de rôle*. Pour chaque *port de rôle* qu'il possède, un *composant connecteur* agrège un *composant rôle* associé. Chaque *port asynchrone de connecteur* exporte un *port asychrone* d'un *composant rôle* contenu. L'export d'un *port asynchrone*

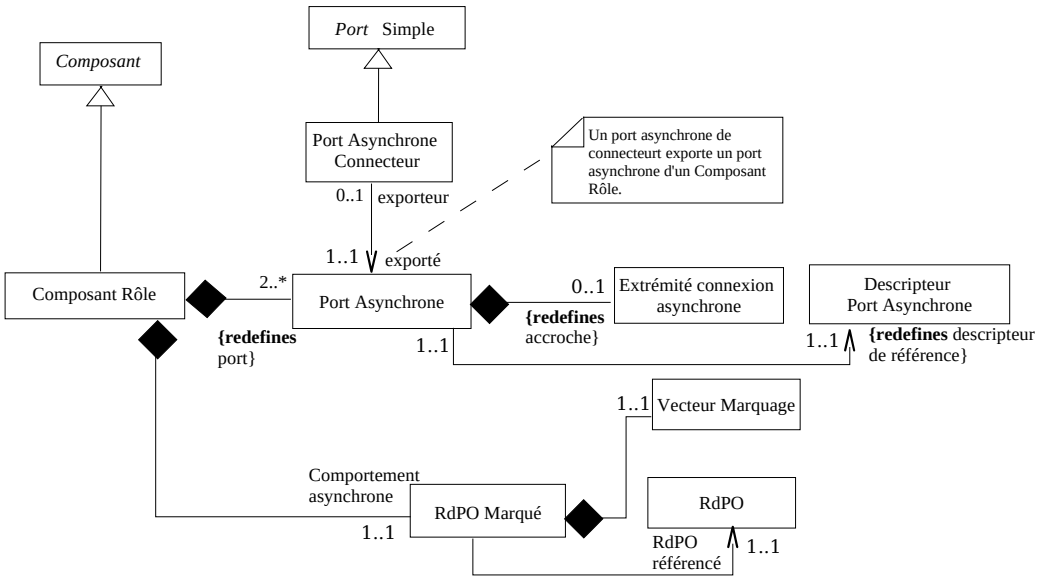


FIG. 7.20 – Meta-modèle d'un composant rôle

signifie qu'à l'exécution, tous les échanges entre un *port asynchrone de connecteur* P1 et un *port asynchrone* P2 d'un *composant de contrôle*, seront traduits sous la forme d'échanges entre P2 et le *port asynchrone* exporté par P1. Chaque *composant rôle* possède également des *ports asynchrones* qui ne seront pas exportés, car ils correspondent aux échanges de messages internes à un *composant connecteur*.

7.4.3 Exemple de syntaxe

L'instanciation du meta-modèle permet de créer des *connecteurs* et leurs descripteurs. Un exemple de *descripteur de connecteur* est donné dans une syntaxe textuelle (cf. fig 7.22, fig. 7.23 et fig. 7.24) et un exemple de *composant connecteur* instance de ce descripteur, est présenté dans une syntaxe graphique (cf. fig 7.25).

Le descripteur `RequestReplyConnexion` des figures 7.22, 7.23 et 7.24 permet de présenter la structure typique d'un *descripteur de composant connecteur*. Il décrit un protocole via lequel un *composant de contrôle* peut jouer un *rôle* `Requester` (envoi de requête) ou un *rôle* `Replyer` (calcul et émission de la réponse). Lorsqu'un composant joue le *rôle* `Requester`, il peut émettre un message et se mettre en attente d'un message en réponse. Il ne pourra émettre un nouveau message que lorsqu'il aura reçu le message de réponse. Lorqu'un composant joue le *rôle* `Replyer`, il est en attente de messages provenant de `Requester`. Lorsqu'il reçoit un message, il le traite et se bloque (ne traite pas de nouveaux messages éventuels) jusqu'à ce qu'il ait émis un message de réponse. Un seul composant peut jouer le *rôle* `Replyer` et un nombre quelconque de *composants de contrôle* peuvent jouer le *rôle* `Requester`. Ce *descripteur de connecteur* est décrit suivant quatre parties contenant les *descripteurs de composant rôle*, la description des ports, des attributs et des opérations.

La première partie (notée `roles definition`) contient la définition de tous les *descripteurs de composant rôle* (cf. fig. 7.22 et 7.23). Les *descripteurs de composant rôle* sont définis à l'intérieur du

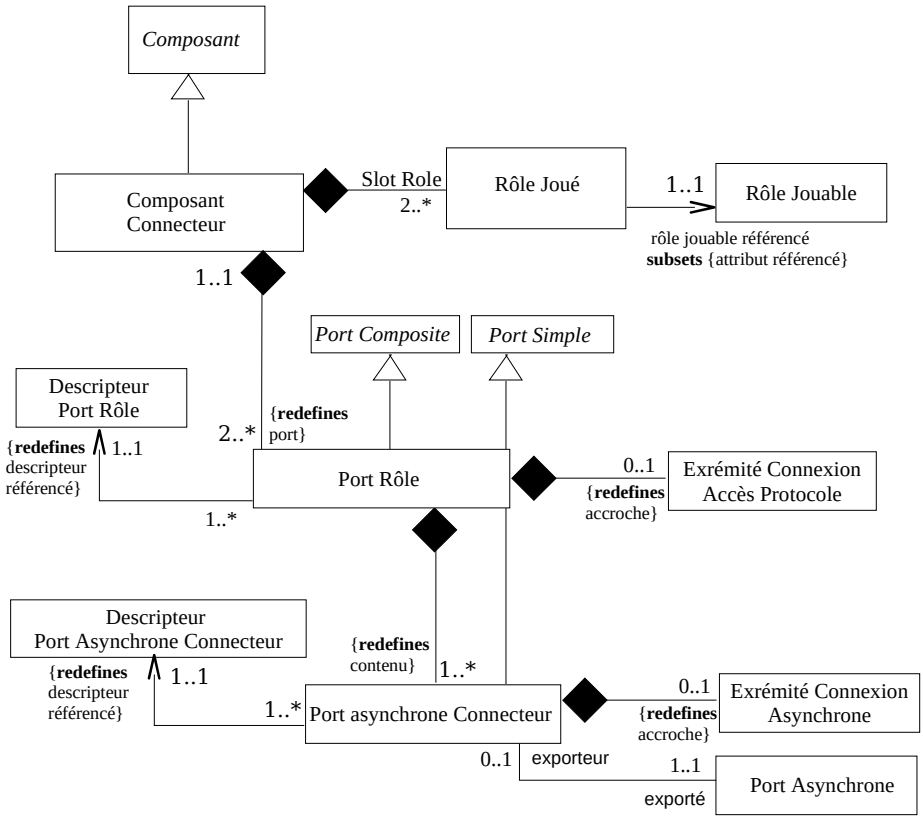
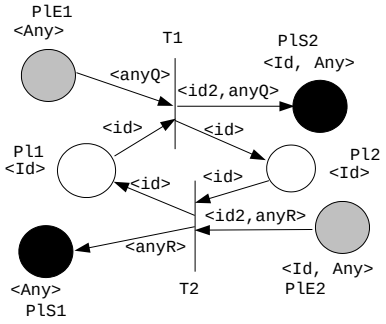


FIG. 7.21 – Meta-modèle d'un connecteur

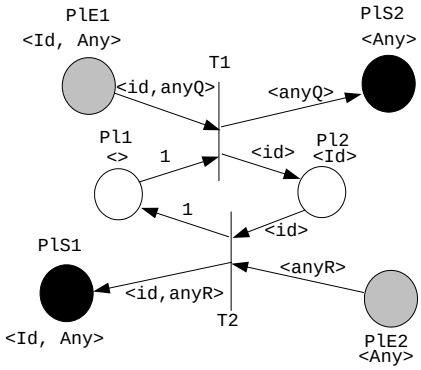
```
Connector Descriptor RequetReplyConnexion <anyQ, anyR>
{
//première partie
roles definition:
import interface Requester<anyR, anyQ>, Replyer<anyR, anyQ>;

Rôle Descriptor RequesterRole <anyQ, anyR>{
ports:
provided asynchronous port Preq1 refers {In sendRequest (Any anyQ)} : PlE1;
provided asynchronous port Preq2 refers {Out receiveReply (Any anyR)} : PlS1;
required asynchronous port Preq3 refers {In transmitReply(Id, Any anyR)} : PlE2;
required asynchronous port Preq4 refers {Out transmitRequest(Id, Any anyQ)} : PlS2;

Attributes:
Id _id;
Operations:

init() //constructeur
{
  _id = create Id();

  //initialisation de la place initialement marquée
  addToken Pl1 <_id>;
}
//destructeur
end()
{
 //vidage du PNO
 removeTokens();
}

Asynchronous Behaviour :
```




```
T1:
action : ID id2 = create ID(id);

T2:
action : destroy id2;

};
//... fin première partie
```

FIG. 7.22 – Exemple d'un descripteur de connecteur : descripteur de composant rôle Requester

```
// deuxième partie

Rôle Descriptor ReplyerRole <anyQ, anyR>{
ports:
provided asynchronous port Prep1 refers {In transmitRequest (<Id,Any anyQ)} : PlE1;
provided asynchronous port Prep2 refers {Out receiveRequest (Any anyQ)} : PlS2;
required asynchronous port Prep3 refers {In sendReply (Any anyR)} : PLE2;
required asynchronous port Prep4 refers {Out transmitReply (<Id, Any anyR)} : PlS1;

Attributes:
//...

Operations:

init() //constructeur
{
 //initialisation de la place intialement marquée
 addToken Pl1 <>;//ajour d'un jeton banalisé à la place Pl1
}

end() //destructeur
{
 //vidage du PNO
 removeTokens();
}

Asynchronous Behaviour:
```




```
T1:
action : //action vide

T2:
action : //action vide

};
//... fin deuxième partie
```

FIG. 7.23 – Exemple d'un descripteur de connecteur : descripteur de composant rôle Replyer

```
//dernière partie

Ports:
  provided role port PR1 refers {Requester<anyR, anyQ>} cardinality : role_requester[0..*]{
  asynchronous port PAs1 refers {In sendRequest(Any anyQ)} : role_requester[].Preq1;
  asynchronous port PAs2 refers {Out receiveReply(Any anyR)} : role_requester[].Preq2;
 }

 required role port PR2 refers {Replyer<anyR, anyQ>} cardinality : role_replyer[1..1]{
  asynchronous port PAs3 refers {Out receiveRequest(Any anyQ)}} : role_replyer.Prep2;
  asynchronous port PAs4 refers {In sendReply(Any anyR)}} : role_replyer.Prep3;
 }

Attributes:
//roles jouables
RequesterRole<anyQ,anyR> [] role_requester;
//[] indique que le connecteur accepte une collection de composants roles requester.
ReplyerRole<anyQ,anyR> role_replyer;

//connesions entre roles jouables
AsynchConnection connexion_OnRequest; //connexions internes au connecteur
AsynchConnection connexion_OnReply;

Operations:

init()//constructeur
{
  //création des composants roles
  for (int i =0; i < numberOfRolePlayer PR1 ; i++)
  {
    role_requester.add(create RequesterRole<anyQ, anyR>);//ajout d'un rôle joué
    attach role_requester[i] with PR1[i];//association port de rôle / composant rôle
  }
  role_replyer = create ReplyerRole<anyQ,anyR>();//un seul rôle Replyer
  attach role_replyer with PR2;

  //création des connexions internes au connecteur
  connexion_OnRequest = connect role_replyer.Prep1 as receiver, role_requester[].Preq1 as emitter;
  connexion_OnReply = connect role_replyer.Prep4 as emitter, role_requester[].Preq3 as receiver;
}

end() //destructeur
{
  //destruction des connexions asynchrones entre les roles
  disconnect connexion_OnRequest;
  disconnect connexion_OnReply;
  //destruction des roles
  destroy [] role_requester;
  destroy role_replyer;
}
};
```

FIG. 7.24 – Exemple d'un descripteur de connecteur : description globale

descripteur de connecteur car leurs définitions sont intrinsèquement couplées entre elles et à celle du *descripteur de connecteur* lui-même. Le descripteur `RequestReplyConnexion` contient la définition de deux *descripteurs de composant rôle* : `RequesterRole <anyQ, anyR>` et `ReplyerRole<anyQ, anyR>`. La définition des *descripteurs de composant rôle* est très proche de celle des *composants de contrôle* : ils contiennent des attributs, des opérations et un comportement asynchrone. Les *descripteurs de port asynchrone* (mot-clef `asynchronous port`) sont décrits dans la partie `ports`. Chaque *descripteur de port asynchrone* référence une interface asynchrone et il est associé à une place d'entrée ou de sortie de son comportement asynchrone, en fonction qu'il est un port d'entrée ou de sortie de messages (correspondant respectivement au qualificatif `In` ou `Out` de l'interface qu'il référence). Par exemple le *descripteur de port asynchrone* `Preq1` du descripteur `RequesterRole` référence l'*interface asynchrone* `sendRequest (Any anyQ)`. `Preq1` est associé à la place d'entrée `PlE1` de son comportement asynchrone.

Dans la partie de description des ports (cf. fig. 7.24), `RequestReplyConnexion` contient deux *descripteurs de port de rôle*, notés `PR1` et `PR2`. Le mot-clef utilisé pour les déclarer est `role port`. Chaque *descripteur de port de rôle* référence une *interface de rôle* (mot-clef `refers`), par exemple `PR1` fait référence à l'interface `Requester<anyR, anyQ>` (décrite plus en détail dans la figure 7.25). Un *descripteur de port de rôle* possède une cardinalité (définie avec le mot clef `cardinality`) qui est associée à un *rôle jouable* : elle définit une contrainte sur le nombre de *composants rôles* instances accessibles via le *rôle jouable* correspondant. Chaque *descripteur de port de rôle* est composé d'un ou plusieurs *descripteurs de port asynchrone de connecteur* (déclarés via le mot clef `asynchronous port`). Par exemple, le *descripteur de port de role* `PR1` est composé de deux *descripteurs de port asynchrone de connecteur* : `PAs1` et `PAs2`. Chaque *descripteur de port asynchrone de connecteur* fait référence à un *interface asynchrone*. Par exemple, `PAs1` fait référence à `sendRequest(Any anyQ)` qui est contenue dans `Requester<anyR, anyQ>` (cf. fig. 7.25). Chaque *descripteur de port asynchrone de connecteur* exporte un ou plusieurs *ports asynchrones* d'un des *rôles jouables*. Par exemple, `PAs1` exporte l'ensemble des ports `Preq1` des *composants rôles* de type `RequesterRole` référencés par le *rôle jouable* `role_requester`(cf. `role_requester[].Preq1`).

Dans la partie de description des attributs, le *descripteur de connecteur* déclare un ensemble de *rôles jouables*. L'attribut `role_requester` représente l'ensemble des rôles `RequesterRole` et l'attribut `role_replyer` représente l'unique rôle `ReplyerRole`. Dans cette même partie, il existe des attributs de type *connexion asynchrone simple* permettant l'assemblage des différents *composants rôles* du *connecteur*.

Dans la partie de description des opérations, les opérations `init` et `end` permettent de gérer la connexion des *composants rôles* agrégés par le *composant connecteur*. La méthode `init` sert à instancier les *composants rôles* et à connecter leurs *ports asynchrones* avec des *connexions asynchrones simples*. Pour ce faire de nouveaux mot-clefs sont proposés :

- `numberOfRolePlayer` *descripteur de port de rôle*, permet de connaître, pour un *descripteur de port de rôle* donné, le nombre de *ports de rôle* instances. Ceci permet à un *composant connecteur* de retrouver le nombre de fois où chaque *rôle* est joué par des *composants de contrôle*.
- `attach` *composant rôle* `with` *port de rôle*, permet d'exprimer une relation entre un *composant rôle* et un *port de rôle*.

L'ensemble de ces mot-clefs permet de configurer dynamiquement la structure interne d'un *composant connecteur* en fonction des *composants de contrôle* participants. La figure 7.25 montre une instanciation partielle d'un *composant connecteur* instance du descripteur `RequestReplyConnexion`. Dans cet exemple, le nombre de *rôles joués* a été défini en fonction du nombre de participants à l'interaction, qui jouent le rôle `Requester` (le *rôle* `Replyer` n'étant joué qu'une fois). La figure 7.26 montre une instanciation partielle de chaque *descripteur de composant rôle*.

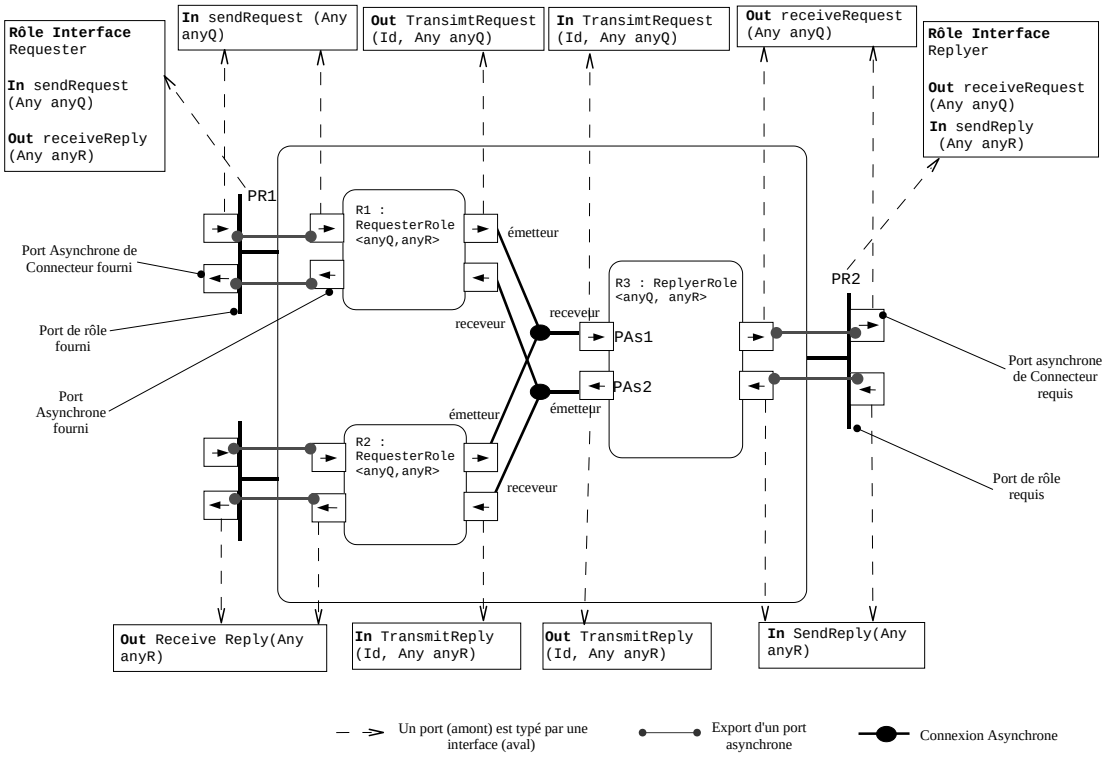


FIG. 7.25 – Exemple d'une instanciation partielle du descripteur de connecteur

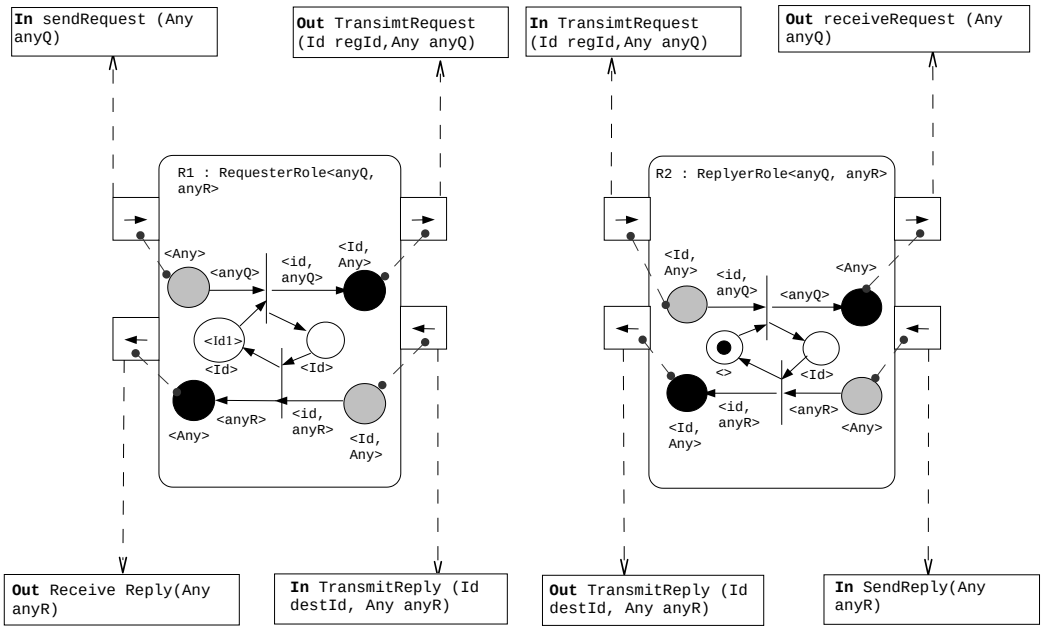


FIG. 7.26 – Exemple interne des rôles instances (paramètres non fixés)

Les *descripteurs de composant connecteur* et les *descripteurs de composant rôle* peuvent être composants paramétrables. C'est le cas dès qu'une de ces entités est définie à partir d'une ou plusieurs variables de type `Any`. Ces variables peuvent être des attributs, des arguments d'opérations ou des variables associées à des interfaces simples ou des variables associées à un RdPO. Lorsqu'une variable est de type `Any`, son type réel doit être fixé au moment de l'instanciation du descripteur correspondant (le type réel n'est jamais fixé pour une interface) : elle est un *paramètre* du descripteur. A l'instanciation, le type de la variable peut être un type primitif, un type de *composant de représentation* ou une liste de types de *composants de représentation*. Le *descripteur de connecteur* et les *descripteurs de composant rôle* présentés dans cette section sont des exemples de descripteurs paramétrables. Par exemple, `RequestReplyConnexion` a pour paramètres `anyQ` et `anyR`, qui permettent de fixer les types des données échangées respectivement lors de l'envoi de la requête et lors de l'envoi de la réponse. Ce mécanisme est particulièrement important pour la description de *descripteurs de composant connecteur* fortement réutilisables et adaptables. En effet, les types des données échangées ne sont pas toujours prépondérants pour la définition d'un protocole d'interaction. Par exemple, le protocole d'interaction défini dans `RequestReplyConnexion` n'est pas dépendant des types des données échangées. Une contrainte à respecter est que le type des données émises (`sendRequest`) par un `RequesterRole` soit le même que celui des données reçues (`receiveRequest`) par le `ReplyerRole`. Inversément, l'autre contrainte à respecter est que le type des données reçues (`receiveResponse`) par un `RequesterRole` soit le même que celui des données émises (`sendResponse`) par le `ReplyerRole`. L'utilisation des paramètres `anyQ` et `anyR` permet d'adapter le protocole en fonction des types des données respectivement émises et reçues par un `RequesterRole`. Il permet également d'assurer que les contraintes sur les types des données échangées, soient respectées entre tous les `RequesterRole` et le `ReplyerRole`. Il faut noter que l'utilisation de paramètres dans les *descripteurs de composant connecteur* n'est pas obligatoire, car dans certains cas, les données échangées et le protocole sont intrinsèquement liés.

Les *descripteurs de composant connecteur* et les *descripteurs de composant rôle* peuvent être basés sur un type primitif spécifique appelé `Id` qui représente le type des identifiants des composants (i.e. une entité possédant des informations d'identification d'un composant dans une architecture logicielle). L'utilisation des `Id` sert à router les messages lorsqu'un émetteur est connecté à plusieurs receveurs dans une *connexion asynchrone simple* : il permet d'identifier le receveur d'un message. Par exemple, le *port asynchrone* `PAs2` est connecté à plusieurs *ports asynchrones* re réception de message. En fonction de l'`Id` associé à un message émis par `PAs2`, ce message sera routé vers le *port asynchrone* du *composant rôle* identifié par l'`Id`. L'`Id` associé au message a évidemment été transmis préalablement au *composant rôle* qui émet le message.

7.5 Assemblage des composants de contrôle

La figure 7.27 montre un exemple d'un *connecteur* utilisé pour l'assemblage de deux *composants de contrôle*. Dans cet exemple, le port `P1` du composant de contrôle 1, est connecté au port `P2` du composant de contrôle 2, via le *connecteur* `Conn1` de descripteur `RequestReplyConnexion`.

La première contrainte, qui est respectée, est que `P1` et `P2` soient typés par des *interfaces de contrôle* compatibles (on remarque que les deux interfaces sont différentes), c'est-à-dire qu'elles portent le même nom de service (`DistanceTo`). Chaque *interface de contrôle* a un rôle associé défini par une *interface de rôle*. Les *interfaces asynchrones* qu'elles contiennent sont définies en fonction des *interfaces asynchrones* contenues dans ces *interfaces de rôle*. Par exemple l'*interface de contrôle* typant le port `P2` a été définie en fonction de l'*interface de rôle* `Replyer`. Ceci indique que le *port de rôle* requis du *connecteur* utilisé pour la connexion doit être typé par l'*interface de rôle* `Replyer`, ce qui est le cas du port `PR2`. L'interface typant `P1` a pour rôle associé celui défini par l'*interface de rôle* `Requester`, ce qui indique que le *connecteur* utilisé doit avoir un port fourni typé par l'*interface de*

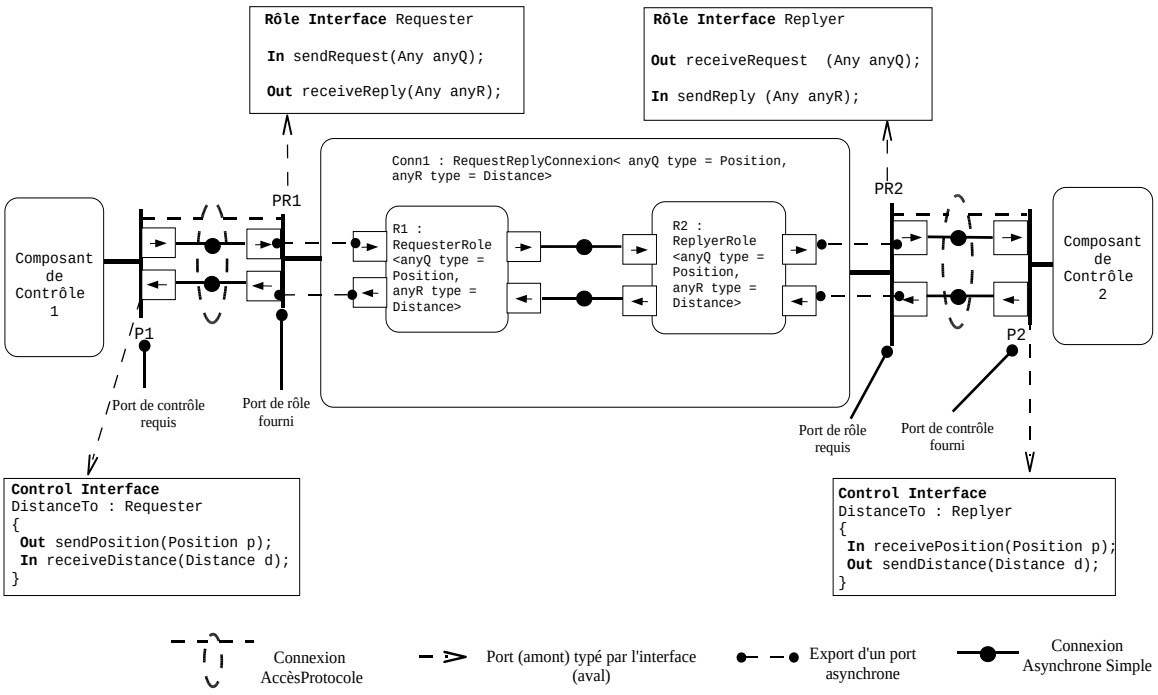


FIG. 7.27 – Exemple d'un connecteur RequestReplyConnexion connectant un port de contrôle fourni et un port de contrôle requis

rôle `Requester`, ce qui est le cas du port `PR1`. Puisque toutes les contraintes ont été respectées (y compris les contraintes de cardinalité du connecteur), une *connexion de contrôle* utilisant le connecteur `Conn1` est possible entre `P1` et `P2`. Notons qu'un *connecteur* instance d'un autre descripteur que `RequestReplyConnexion` aurait pu être utilisé si un de ses *ports de rôle* fournis était typé par l'interface `Requester` et qu'un de ses *ports de rôle* requis était typé par l'interface `Replyer`. Le *connecteur* `Conn1` et ses *composants rôles* ont été paramétrés en fonction des types des paramètres utilisés pour la connexion des ports des deux *composants de contrôle*. Ainsi, le paramètre `anyQ` a pour type `Position` et le paramètre `anyR` a pour type `Distance`.

Pour qu'une *interface de contrôle* soit compatible avec une *interface de rôle*, elle doit définir une *interface asynchrone* compatible pour chaque *interface asynchrone* de l'*interface de rôle*. La compatibilité est établie en fonction de deux critères :

- les deux *interfaces asynchrones* ont une qualité différente, l'une est *In* et l'autre est *Out* (e.g. `sendPosition` et `sendRequest`).
- les paramètres d'une *interface asynchrone* contenue dans l'*interface de contrôle*, doivent être compatibles avec ceux définis dans l'*interface asynchrone* contenu dans l'*interface de rôle*. Cette compatibilité s'établit simplement en fonction des types des paramètres, en prenant en compte que le type `Any` peut être remplacé par n'importe quelle liste de types (qui peut éventuellement être vide).

Notons que, pour simplifier, nous considérons que les *interfaces asynchrones* sont ordonnées au sein des *interfaces de rôle* et des *interfaces de contrôle*) et que les paramètres sont sont ordonnés au sein des *interfaces asynchrones*.

L'assemblage des *composants de contrôle* nécessite des connexions de ports à différents niveaux de granularité. Des *connexions d'accès de protocole* (cf. fig. 7.15) sont établies entre les *ports de*

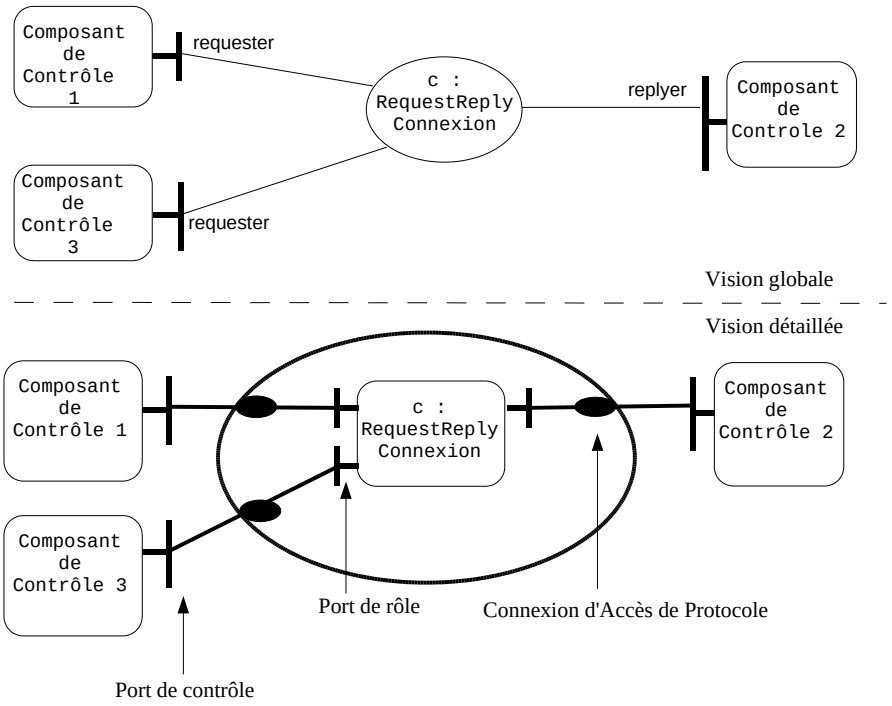


FIG. 7.28 – Représentation graphique d'une connexion de contrôle

contrôle et les *ports de rôle*. Elle sont composées de *connexions asynchrones* entre les *ports asynchrones* contenus dans les *ports de contrôle* et ceux contenus dans les *ports de rôle*. Une *connexion de contrôle* est composée d'autant de *connexions d'accès de protocole* qu'il y a de *ports de contrôles* connectés. Elle est également composée d'un unique *connecteur*. Néanmoins cette complexité n'est pas forcément nécessaire au moment de la lecture du modèle, c'est pourquoi nous proposons deux façons de représenter les *connexions de contrôle*, une globale et une autre qui détaille la connectique des ports du *composant connecteur* utilisé. Dans ces représentations, les *connexions asynchrones* ne sont pas représentées (elle peuvent être déduites des *connexions d'accès de protocole*) ce qui permet de simplifier la lecture.

Malgré sa complexité, le mécanisme de connexion est puissant dans le sens où, grâce à la notion de *connecteur*, il est possible de définir des *connexions de contrôle* n-aires (plus de deux extrémités) et basées sur un nombre quelconque de rôles potentiellement jouables. Ce mécanisme de connexion, amène la possibilité de reconstituer le comportement asynchrone global d'un assemblage de composants (cf. fig. 7.29). Ce comportement asynchrone global est issu de la composition des comportements asynchrones des *composants de contrôle* avec les comportements asynchrones des *composants rôles*, ainsi que de la composition des *composants rôles* entre eux (cf. fig. 7.29). Le mécanisme de composition des RdPO est basé sur le mécanisme de fusion des places d'entrée et de sortie de messages. Une fusion est créée lorsqu'au moins une place de sortie et au moins une place d'entrée sont fusionnée afin de créer une place interne. Un nombre quelconque de places d'entrées et de sorties peut être utilisé dans une fusion, mais une même place ne peut être concernée que dans une unique fusion. La place interne, créée par fusion, a pour arcs amonts l'ensemble des arcs amonts à toutes les places de sortie fusionnées et a pour arcs avals l'ensemble des arcs avals à toutes les places d'entrée fusionnées.

Les fusions de places sont intégralement spécifiées dans la description des *connexions asychrones*

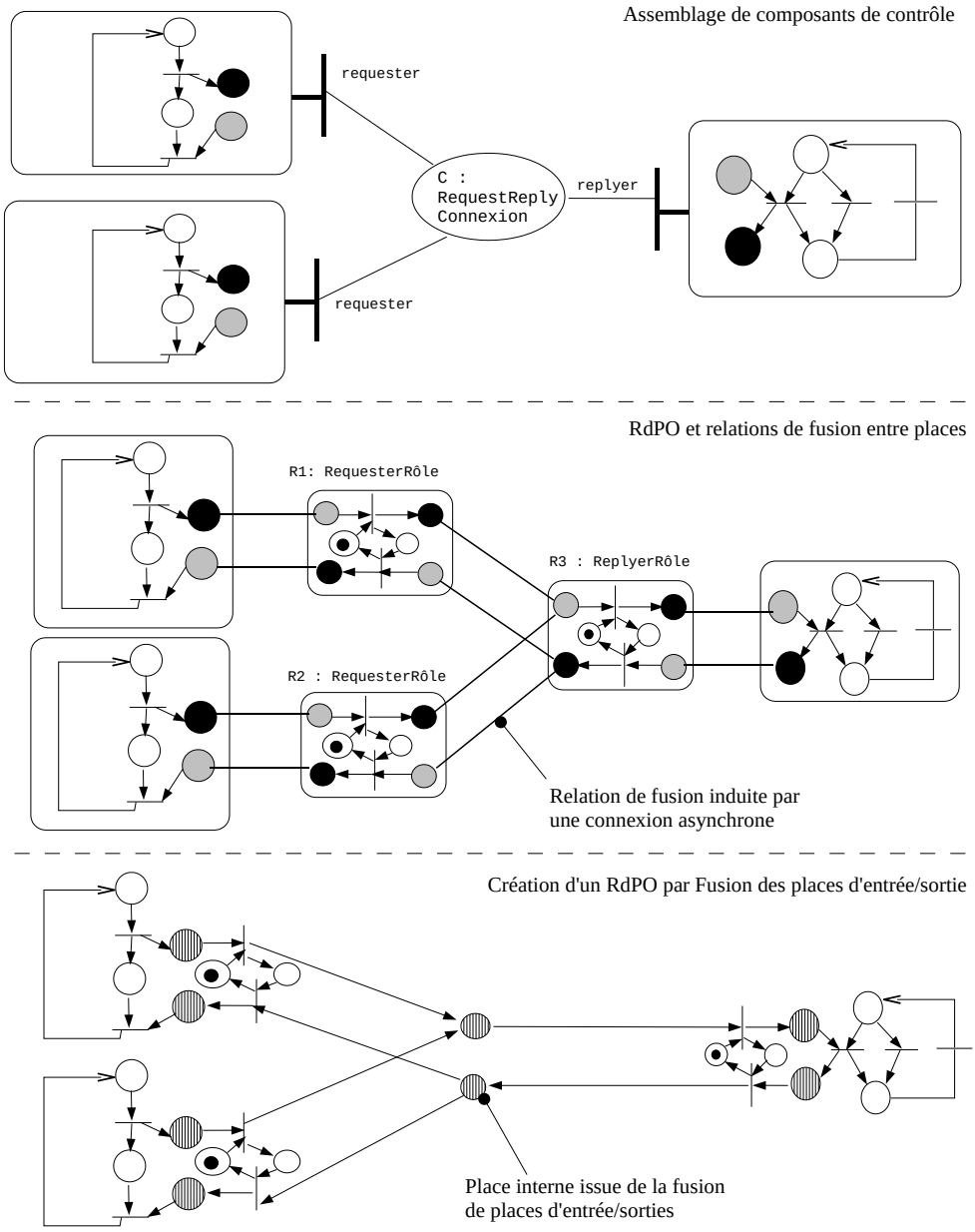


FIG. 7.29 – Création d'un RdPO par fusion des comportements asynchrones

entre les *ports asynchrones* des composants. Chaque *port asynchrone* est relié soit à une unique place d'entrée soit à une unique place de sortie d'un comportement asynchrone. Réciproquement, chaque place d'entrée et chaque place de sortie est reliée à un unique port asynchrone d'un composant. Une *connexion asychrone* entre des *ports asynchrones* permet de définir une relation de fusion entre les places d'entrée et les places de sortie correspondantes. Ce mécanisme permet de formaliser la composition de composants de contrôle, ce qui peut permettre, ultérieurement, d'analyser le résultat d'une composition de comportements asynchrones.

7.6 Configurations

7.6.1 Introduction

La dernière catégorie de composants proposée par le langage COSARC est celle des *composants de configuration*, plus simplement appelés *configurations* (par analogie avec les ADL [86]). Les *configurations* sont des composants qui agrègent une architecture de contrôle, c'est-à-dire des assemblages de composants de contrôle. L'intérêt de cette catégorie de composants est d'abstraire et de réutiliser une architecture logicielle sous la forme d'un composant, ce qui permet de décrire l'architecture de contrôle suivant une approche systémique. Les *configurations* sont utilisées pour représenter des système et sous-systèmes (i.e. les entités de contrôle composites), telles que les ressources ou les contrôleurs de robots (cf. fig. 6.2). D'autre part, les *configurations* sont les entités à partir desquelles est décrit le déploiement des architectures logicielles qu'elles agrègent.

Le déploiement est une phase particulièrement importante dans le monde des composants logiciels. Elle consiste à mettre en correspondance les modèles ayant permis la description logique d'une architecture logicielle d'une application avec le monde physique réel dans lequel cette application va s'exécuter. Carzaniga [38] définit le déploiement de la façon suivante : *the term software deployment refers to all activities that make a software system available for use.* Le processus de déploiement d'une application logicielle à base de composants peut être décomposé en plusieurs activités [38] [66], qui varient en fonction des technologies et modèles de composants [70]. Dans le cadre du langage CoSARC, nous limitons le déploiement à l'ensemble des activités suivantes :

- l'activité de *préparation* permet de préparer le déploiement d'une application logicielle. Cela consiste à décrire le placement des composants sur une représentation de l'infrastructure de déploiement dans laquelle les composants vont s'exécuter. L'infrastructure de déploiement en elle-même est vue comme un système matériel (les nœuds et liens de communication entre noeuds) et logiciel (OS et middleware).
- l'activité d'*installation* consiste à installer le code de définition des composants d'une architecture logicielle sur une infrastructure matérielle cible. Cela passe par une diffusion du code des descripteurs et des composants, puis par la configuration de l'infrastructure de déploiement (e.g. enregistrement des descripteurs et composants dans des référentiels). L'activité de *désinstallation* consiste à effectuer l'opération inverse, afin qu'une infrastructure matérielle ne soit plus capable d'exécuter une application logicielle.
- l'activité d'*activation* (ou *lancement*) permet d'amener l'application logicielle dans un état d'exécution. L'*activation* consiste à créer les composants et à les connecter, puis à démarrer leur exécution. La *désactivation* (ou *arrêt*) permet de réaliser l'opération inverse.

Dans le langage CoSARC, l'activité de *préparation* est décrite par les développeurs via des notations spécifiques. L'environnement d'exécution du langage CoSARC est en charge de la phase d'*installation* : celle-ci est réalisée en fonction de la description du déploiement réalisée au cours de la phase de *préparation*. De façon idéale, l'environnement d'édition diffuse le code logiciel vers l'environnement d'exécution (middleware) qui se charge du reste des opérations liées à l'*installation*. Enfin, l'environnement d'exécution se charge de l'*activation* et de la *désactivation* de l'application,

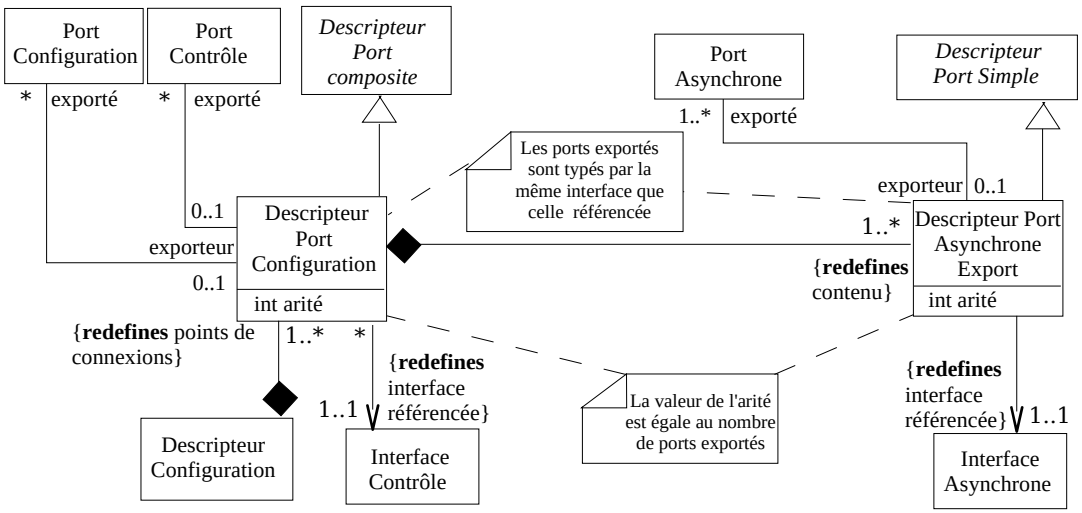


FIG. 7.30 – Meta-modèle d'un descripteur de configuration

en fonction de consignes provenant de l'environnement d'édition (et donc indirectement des développeurs). L'essentiel du travail des développeurs consiste donc à décrire le déploiement des architectures logicielles sur des infrastructures de déploiement. L'environnement d'exécution CoSARC doit automatiser (au moins en partie) le reste des activités de déploiement en se basant sur cette description.

Pendant la phase de *préparation*, la description du déploiement est réalisée via les *composants de configuration*. Elle peut être découpée en trois phases :

- la description de l'architecture logicielle d'un contrôleur, sans prendre en compte les détails relevant de son déploiement. Elle est réalisée via la définition de *descripteurs de configuration*.
- la description de l'infrastructure de déploiement (*Target Environnement* dans le jargon de l'OMG) sur laquelle sera déployée l'architecture logicielle du contrôleur.
- la description du *placement* (*Execution* dans le jargon de l'OMG) des composants de l'architecture logicielle sur l'infrastructure de déploiement. C'est à cette étape que l'architecture logicielle est adaptée au contexte matériel/système/réseau qui lui permet de s'exécuter. C'est également à cette phase que sont configurées les propriétés non-fonctionnelles des composants.

7.6.2 Meta-modèle

Un *descripteur de composant de configuration* (cf. fig. 7.30) contient un ensemble de *descripteurs de port de configuration* qui donneront chacun naissance à un unique *port de configuration*. Un *descripteur de port de configuration* référence une *interface de contrôle* ; il exporte un ensemble de *ports de contrôle* qui appartiennent à des *composants de contrôle* attributs du *descripteur de configuration* ; il contient un ensemble de *descripteurs de port asychrone d'export*. Chaque *descripteurs de port asychrone d'export* exporte un ou plusieurs *ports de contrôle* appartenant à un ou plusieurs des *composants de contrôle* contenus et/ou un ou plusieurs *ports de configuration* appartenant à un ou plusieurs des *composants de configuration* contenus. La sémantique de l'export des *ports de contrôle* et des *ports asynchrones* est la suivante : toute connexion entre un port exporteur P1 et un port P2 sera traduite à l'exécution en une connexion établie entre tous les ports exportés par P1, et P2. A chacun de ces descripteurs de port est associée une *arité*, qui correspond au nombre de ports qu'ils exportent. Un *descripteur de configuration* définit des attributs dont le type est défini par

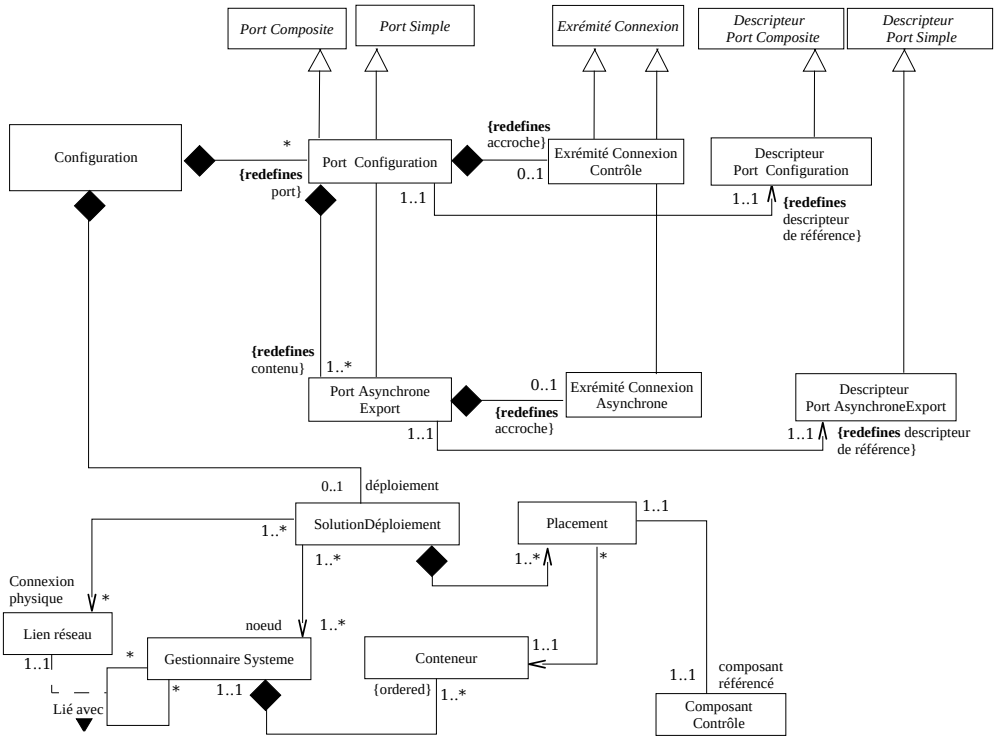


FIG. 7.31 – Meta-modèle d'une configuration

des descripteurs de composants. Certains de ses attributs ont des types définis par des *descripteurs de composant de contrôle* et des *descripteurs de connecteur*, voire des *descripteurs de configuration*. Ces attributs permettent de représenter l'architecture de contrôle contenue dans un *descripteur de configuration*. Les attributs dont le type est primitif ou défini par un *descripteur de composant de représentation* permettent de représenter les connaissances communes à tous les *composants de contrôle*, *connecteurs* et *configurations* contenus. Par exemple, la ressource `Mobile` possède un attribut qui représente la connaissance du `Véhicule` contrôlé. Un *descripteur de configuration* possède une opération d'initialisation et une opération de fin qui permettent de construire et détruire l'architecture de contrôle interne à une *configuration* instance.

Un *composant de configuration*, instance d'un *descripteur de configuration*, agrège un ensemble de *composants de contrôle*, de *connecteurs* et de *connexions de contrôle* qui correspondent à ses attributs. Au moment de la création d'une *configuration*, chaque *descripteur de port* est instancié une unique fois afin de créer un port correspondant. Les interfaces typant les *ports de configuration* étant les mêmes que celles typant les *ports de contrôle*, il est possible de considérer, à un niveau de détail faible, une *configuration* comme un *composant de contrôle*. Ainsi, il est possible de connecter une *configuration*, indifféremment avec un *composant de contrôle* ou une autre *configuration*. Les règles s'appliquant à la définition des *ports de configuration* sont les suivantes :

- Un *port de configuration* exporte un ensemble de *ports de contrôle* et/ou *ports de configuration* dont le nombre est défini par l'arité de son descripteur. Chacun de ces ports est typé par la même *interface de contrôle*.
- Un *port asynchrone d'export* exporte un ensemble de *ports asynchrones* dont le nombre est défini par l'arité de son descripteur. Chacun de ces *ports asynchrones* est typé par la même *interface asynchrone*. L'arité d'un *descripteur de port asynchrone d'export* est égale à l'arité du

descripteur de port de configuration qui le contient.
La règle imposée pour la connexion d'un *port de configuration* est que le *port de rôle* (du *connecteur* utilisé) doit avoir une cardinalité compatible avec l'arité du *port de configuration* : l'arité doit être comprise entre les bornes inférieures et supérieures de la cardinalité.

Une *configuration* a la particularité de pouvoir se déployer selon une *solution de déploiement*. Une *solution de déploiement* contient la description du placement des *composants de contrôle* sur une infrastructure de déploiement. La *solution de déploiement* précise les sites physiques, hôtes d'environnement d'exécution de composants, ainsi que les connexions physiques entre ces sites. Un site est représenté par une entité nommée *gestionnaire de système*, qui contient un ensemble ordonné de *conteneurs*. Un *conteneur* est une unité d'exécution et de mémoire (i.e. assimilable à un processus) pour un ensemble de *composants de contrôle* et de *composants rôles*. Tous les *conteneurs* présents sur un même *gestionnaire de système* sont ordonnés en fonction de leur priorité d'exécution relative. Les *gestionnaires de systèmes* sont reliés entre eux via des *liens* qui représentent les réseaux de communication physiques. La solution de déploiement contient un ensemble de *placements*. Un *placement* est une information qui met en correspondance un *composant de contrôle* et un *conteneur* dans lequel ce composant s'exécute.

7.6.3 Exemple de syntaxe

Un exemple textuel de *descripteur de configuration*, conforme au meta-modèle présenté, est donné dans la figure 7.32. Un exemple graphique d'une *configuration* instance de ce descripteur est donné dans la figure 7.33. Un *descripteur de configuration* est décrit suivant trois parties : la définition des ports, la définition de ses attributs et la définition de ses opérations.

La partie de définition des ports contient la définition des *descripteurs de port de configuration* (notés `configuration ports`). Chaque *descripteur de port de configuration* référence une *interface de contrôle* et exporte un ou plusieurs *ports de contrôle* et/ou *ports de configuration* qui référencent la même interface. `Pconf1` référence l'interface `IC1` et exporte le port de contrôle `P1`. Chaque *descripteur de port de configuration* est composé d'un ensemble de *descripteurs de port asynchrone d'export* (déclarés via le mot-clef `asynchronous port`). `PConf1` est par exemple composé d'un seul *descripteur de port asynchrone d'export* `PAs1` qui référence l'interface asynchrone `In DoIt(CR4 c4)` qui est aussi référencée par le *port asynchrone* exporté `PAs1` du composant de contrôle `CC1`.

Dans sa partie de définition des attributs, un *descripteur de configuration* déclare un ensemble d'attributs de type *descripteur de composant de contrôle* (e.g. `DescrComp1 CC1`), de type *descripteur de connecteur* (e.g. `RequestReplyConnexion<anyQ,anyR>`) et de type `ControlConnection`. Les attributs de type `ControlConnection` permettent de représenter les *connexions de contrôle* et ils sont construits à partir des attributs dont le type est définit par un *descripteur de connecteur*. Dans la partie de définition des opérations, une configuration contient uniquement les opérations `init` et `end`. L'opération `init` est chargée, à l'instanciation, de générer le graphe des *composants de contrôle* et *composants de configuration* connectés. `init` instancie tout d'abord l'ensemble de ses attributs dont le type est défini par un *descripteur de composant de contrôle*. C'est lors de cette instanciation qu'il peut paramétrer les *composants de contrôle* avec des *composants de représentation*. Par exemple, tous les *composants de contrôle* sont instanciés avec pour paramètre le *composant de représentation* `c1` de type `CR1`. `init` crée ensuite les *connexions de contrôle* en même temps que les *connecteurs*. Pour cela, le mot-clef `connect` est toujours utilisé, mais il est enrichi du mot-clef `using` qui permet de paramétrer la *connexion de contrôle* en fonction d'un *connecteur*. Les *rôles* jouables sont alors eux définis par les *interfaces de rôle* référencées par les *descripteurs de port de rôle* du *connecteur* utilisé. La figure 7.33 présente un schéma d'une instance du *descripteur de configuration* de la figure 7.32. Le graphe de composition des composants de cette *configuration* montre le résultat de la composition

```
Configuration Descriptor ConfigurationExemple
{
ports:
 import interface IC1;
  provided configuration port PConf1 refers {IC1} arity = 1 : CC1.P1 {
   asynchronous port PAs1 refers {In DoIt(CR4)} : CC1.P1.PAs1;
  }

  provided configuration port PConf2 refers {IC2} arity = 1 : CC1.P2 {
     asynchronous port PAs2 refers {In DoItAgain()} : CC1.P3.PAs2;
     asynchronous port PAs3 refers {Out AgainResponse()} : CC1.P3.PAs3;
   }
}
//pas de port de configuration requis sur l'exemple

Attributes:
import descriptor DescrComp1, DescrComp2, DescrComp3, RequestReplyConnexion<anyQ,anyR>;
DescrComp1 CC1; DescrComp2 CC2; DescrComp3 CC3;
RequestReplyConnexion<anyQ,anyR> Conn1, Conn2, Conn3;
ControlConnection connexion1, connexion2, connexion3;

Operations:
import descriptor CR1, CR2, CR3, CR4, CR5, CR6, CR7;

init(CR1 c1) //constructeur
{
 //création des composants de controle contenus avec pour paramètre un composant de représentation
 CC1 = create DescrComp1(C1);
 CC2 = create DescrComp2(C1);
 CC3 = create DescrComp3(C1);

 //création des connexions entre ports de contrôle et instaciation des connecteurs
 connexion1 = connect CC1.P2 as Requester, CC2.P8 as Requester, CC3.P9 as Replyer, using Conn1 = create RequestReply<anyR
type = CR2, anyQ type = CR3>();
 connexion2 = connect CC1.P4 as Requester, CC2.P7 as Replyer, using Conn2 = create RequestReply<anyR type = CR4, anyQ
type = CR5>();
 connexion3 = connect CC1.P5 as Requester, CC2.P6 as Replyer, using Conn3 = create RequestReply<anyR type = CR6, anyQ
type = CR7>();
}

end()//destructeur
{

 //destruction des connexions asynchrones de contrôle entre ports de contrôle contenus et destruction autmatique  des
 //connecteurs

 disconnect connexion1;
 disconnect connexion2;
 disconnect connexion3;

 //destruction des composants de contrôle
 destroy CC1();
 destroy CC2();
 destroy CC3();
}
};
```

FIG. 7.32 – Exemple d'un descripteur de configuration

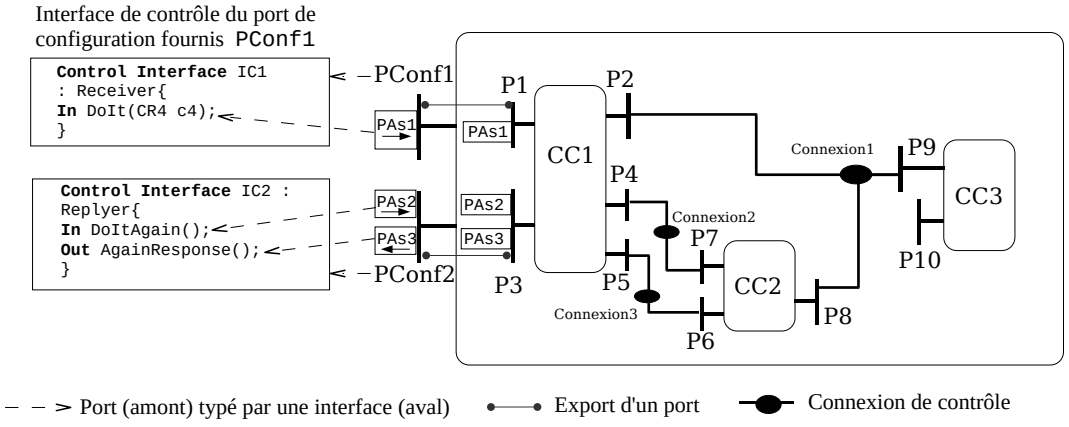


FIG. 7.33 – Exemple d'une configuration instance avant déploiement

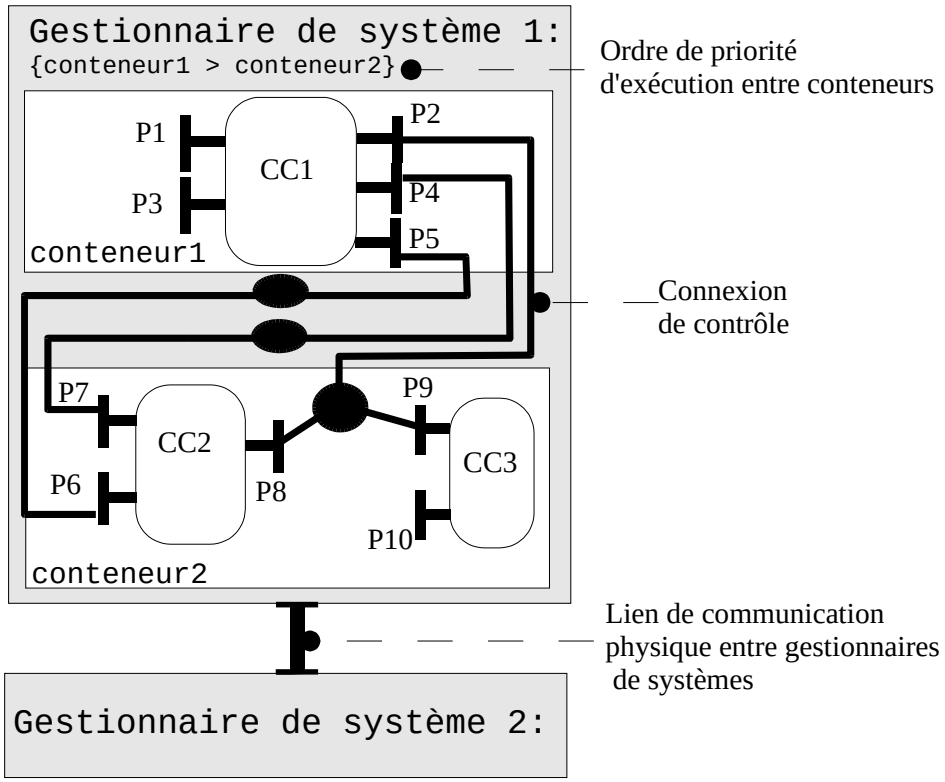


FIG. 7.34 – Description du déploiement d'une configuration

définie au niveau de la méthode `init` de son descripteur. On remarque qu'un *connecteur* est utilisé dans une *connexion de contrôle* qui sert à connecter trois *ports de contrôle* (cas du *connecteur* `Conn1`, utilisé dans la connexion `connexion1`). On remarque qu'on a simplifé la description des *connexions de contrôle* puisqu'à partir des informations décrites on peut déduire toutes les *connexions asynchrones*.

7.6.4 Déploiement

La figure 7.34 présente un schéma simplifié de la *configuration* précédente déployée sur une représentation de l'infrastructure matérielle de déploiement. Chaque *composant de contrôle* est placé dans un *conteneur* et chaque *conteneur* est placé sur un *gestionnaire de système*. La seule propriété non-fonctionnelle actuellement gérée est celle des priorités d'exécution des *conteneurs*. Par exemple, sur le *gestionnaire de système 1* (sur lequel a été déployée l'intégralité de l'architecture de la *configuration*), le *conteneur 1* a une priorité d'exécution supérieure à celle du *conteneur 2*. Ceci implique que les activités des *composants de contrôle* contenus dans le *conteneur 1* pourront préempter les activités des *composants de contrôle* contenus dans le *conteneur 2*.

La description d'une *solution de déploiement* se fait de façon récursive si une *configuration* contient elle-même une ou plusieurs *configurations*. Cette description récursive s'applique à une représentation de l'infrastructure de déploiement globale pour toutes les *configurations*. Des *composants de contrôle* utilisés à différents niveaux d'agrégation peuvent ainsi être placés dans les mêmes *conteneurs* ou sur les mêmes *gestionnaires de système*.

La description du déploiement d'une *configuration* est faite en plusieurs étapes. La première consiste à décrire l'*infrastructure de déploiement* en terme de *gestionnaires de systèmes* interconnectés via des *liens de communication*. La deuxième consiste à décrire le placement des différents *composants de contrôle* dans des *conteneurs*. Le placement des *composants rôles* est entièrement déduit du placement des *composants de contrôle* : un *composant rôle* est exécuté dans le *conteneur* contenant le *composant de contrôle* qui est connecté avec lui via une *connexion d'accès au protocole*. La dernière phase consiste à placer les *conteneurs* sur les *gestionnaires de système*, et à fixer leurs priorités d'exécution relatives (entre les *conteneurs* présents sur un même *gestionnaire de système*).

7.7 Conclusion

Le langage à composants COSARC a été présenté à travers un meta-modèle et des exemples de syntaxe. Son caractère original repose sur plusieurs considérations :

- Le langage repose sur l'utilisation de différentes catégories de composants pour gérer chaque préoccupation de manière indépendante. De plus, le fait que ces entités reposent sur un meta-modèle d'entités abstraites et extensibles permettra dans le futur d'enrichir le langage de nouvelles catégories de composants adaptées à la description de nouveaux aspects (e.g. intégration de services d'accès au matériel, de services système d'ordonnancement, etc.).
- Le modèle d'assemblage présenté reprend les caractéristiques des mécanismes d'assemblage proposés dans les ADLs et les modèles de composants pour fournir une grande flexibilité au niveau des connexions entre composants de contrôle. Le concept de *connecteur* apporte la possibilité d'apdapter les protocoles utilisés en fonction des *composants de contrôle* à assembler et permet de réutiliser les protocoles indépendemment des *composants de contrôle* eux-mêmes. Cette flexibilité rend le langage utilisable pour implanter des architectures de contrôle suivant différents modèles d'organisation (il est évidemment utilisé en premier lieu avec notre propre modèle d'architecture).
- La chaîne de raffinement est globalement simplifiée puisque le langage repose sur des notations adaptées à la description des composants à différents niveaux de détail.

Les éléments de description (du meta-modèle) accessibles en fonction du niveau de détail sont définis en fonction de la catégorie de composant considérée.

- Au niveau de détail 1 (vue structurelle), seuls les ports et les interfaces sont visibles et manipulables, quelle que soit la catégorie de composants.
- Au niveau de détail 2 (vue comportementale), les attributs et les signatures des opérations sont visibles et manipulables. Les opérations sont rattachées aux ports pour les composants de représentation. Pour les composants de contrôle et les rôles, leur comportement asynchrone est également visible et manipulable en plus des attributs et signatures d'opérations. Pour un connecteur, les rôles jouables sont visibles et rattachés aux descripteurs de ports. Pour une configuration, l'architecture contenue est visible et manipulable et l'on peut voir son comportement asycnhrone issu de la composition des comportements asynchrones des composants de contrôle et connecteurs contenus.
- Au niveau de détail 3 (code), le code d'implantation des opération est visible et manipulable, quelle que soit la catégorie de composants.
- Au niveau de détail 4, on décrit le déploiement les configurations, composants de contrôle et connecteurs. Seuls les composants de représentation ne sont pas explicitement déployés.

Ces niveaux de détail sont des abstractions qui peuvent, par exemple, être manipulables au sein de l'environnement d'édition du langage. Un tel environnement s'avère également nécessaire pour permettre aux développeurs de facilement gérer la syntaxe du langage CoSARC, qui reste laborieuse à manipuler sans outils facilitant l'écriture, la recherche des composants et des interfaces et leur visualisation sous forme graphique. Pour ce qui est de l'écriture des opérations, leur code

procédural est basé sur une syntaxe classique (syntaxe C/C++ “simplifiée”). Le développement de l’environnement d’édition du langage CoSARC a démarré dernièrement et n’est pas disponible à l’heure actuelle. Pour cette tâche, le meta-modèle se révèle être plus qu’un élément “cosmétique” de notre proposition : il est utilisé dans une démarche de développement par génération de code à partir de meta-modèles pour produire l’environnement d’édition. Dans le chapitre suivant nous présentons la spécification de l’environnement d’exécution des composants CoSARC.

Résumé :

Le Langage à composants :

- est défini à travers un meta-modèle.

- est utilisé pendant le processus de développement d'une architecture de contrôle et qui permet un raffinement simple à travers les phases de conception, de programmation et de description du déploiement.

- définit des catégories de composants adaptées à la gestion de différentes préoccupations :
  - **composants de représentation**, pour décrire les connaissances que possède le contrôleur sur le monde sur lequel il agit (partie opérative, environnement, etc.).
  - **composants de contrôle**, pour décrire les activités du robot.
  - **composants connecteurs** (ou **connecteurs**), pour décrire les protocoles régissant les interactions entre les composants de contrôle.
  - **composants de configuration** (ou **configurations**), pour décrire des architectures de contrôle et leur déploiement.

- définit deux mécanismes de composition :
  - l'**assemblage** consiste à connecter les ports des composants.
  - l'**agrégation** consiste à créer des composants composites, qui contiennent des composants dont ils dirigent le cycle de vie.

- repose sur l'utilisation des **réseaux de Petri à Objets** pour décrire les activités et les protocoles.

Chapitre 8

Environnement d'exécution

Dans le chapitre précédent, nous avons décrit le langage à composants CoSARC, qui sera utilisé dans un environnement d'édition dédié. Un programme "source" décrit une architecture logicielle de contrôle et son déploiement. La description globale d'un programme "source" comporte l'instanciation du descripteur de configuration qui englobe l'intégralité de l'architecture. Elle comporte également la description de la solution de déploiement de la configuration instance. Cette description est transmise, sous forme de fichier XML, à l'environnement d'exécution, auquel est consacré ce chapitre.

L'environnement d'exécution est une infrastructure matérielle comportant un ou plusieurs sites d'exécution (i.e. nœuds de calcul) reliés par des liens physiques assurant leurs communications (cf. fig. 8.1). Sur chaque site, on trouve un *gestionnaire de système* responsable du déploiement et de l'exécution d'une partie de l'architecture globale. Un *gestionnaire de système* est capable d'interpréter les informations de déploiement contenues dans une configuration, afin de réaliser les opération de déploiement correspondantes. Il fournit un support pour l'exécution et la communication des composants (de représentation, de contrôle et connecteurs) qu'il héberge.

Pour des raisons de simplification, nous n'avons pas, à ce jour, traité le cas du déploiement sur une infrastructure matérielle distribuée. En effet, la distribution pose des problèmes comme la qualité de service dans les communications réseau, qui sont hors du cadre de la thèse. Toutefois, la spécification du *gestionnaire de système* prend en compte cette possibilité future, qui fait partie de nos perspectives.

Ce chapitre présente la spécification globale d'un *gestionnaire de système* CoSARC. Il définit la façon dont un *gestionnaire de système* gère la création et l'ordonnancement des *conteneurs*, ainsi que le déploiement et l'exécution des composants au sein d'un *conteneur*. Nous focalisons la présentation sur le modèle d'exécution et de communication des composants, basé sur l'exécution des RdPO.

8.1 Spécification globale d'un Gestionnaire de système

Un gestionnaire de système CoSARC se présente comme une surcouche d'un système d'exploitation (temps-réel) dont les différentes entités constituantes sont présentées dans la figure 8.1 :

- Le *gestionnaire de communication* est en charge de toutes les communications réseaux autorisées par le site sur lequel il est déployé. Il assure donc les communications avec les autres *gestionnaires de système* de l'environnement d'exécution, dans le cas où l'architecture logicielle est déployée sur une infrastructure matérielle distribuée. La plate-forme opérateur étant considérée comme un *gestionnaire de système*, le *gestionnaire de communication* permet à un opérateur d'interagir avec les composants déployés. Il est également en charge des communications entre le site concerné et l'environnement d'édition des composants CoSARC, ce qui sera utile pour contrôler à distance les opérations de déploiement.

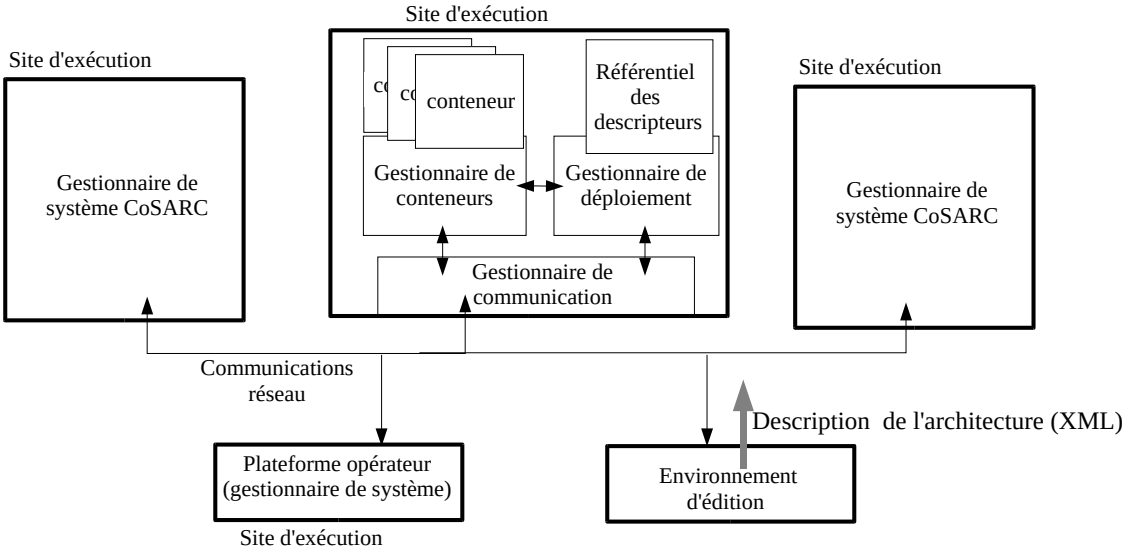


FIG. 8.1 – Schéma global de l'environnement d'exécution CoSARC

- Le *gestionnaire de déploiement* est en charge de toutes les opérations de déploiement sur le site considéré.
- Le *gestionnaire de conteneurs* est en charge de gérer l'ordonnancement des *conteneurs* déployés sur le site. Les *conteneurs*, en eux-mêmes, supportent l'exécution et les communications des composants qu'il contiennent.

La spécification du gestionnaire de communication n'a pas été abordée durant cette thèse. Les aspects "réseau" posent un ensemble de problèmes complexes que nous ne pouvons pas aborder. Cela nécessite une étude préalable conséquente, aussi bien des besoins en terme de qualité de service, que des solutions technologiques adaptées aux contraintes posées par les systèmes "temps-réel". Le framework ACE/TAO [97] est un exemple de solution technologique utilisable pour la gestion des communications réseaux.

8.2 Gestionnaire de déploiement

Le gestionnaire de déploiement est en charge des tâches de déploiement de tout ou partie d'un contrôleur sur un site. Pour cela, il met en œuvre le mécanisme schématisé dans la figure 8.2 qui se déroule en plusieurs étapes :

- La première étape (1) consiste à recevoir du *gestionnaire de communication*, la configuration décrivant l'intégralité d'une architecture logicielle déployée sur l'ensemble de l'infrastructure matérielle. Cette description est contenue dans un fichier XML issu de l'environnement d'édition.
- A partir de cette description globale (2), le *gestionnaire de déploiement* tire de nombreuses informations. Il se positionne par rapport à la globalité de l'infrastructure matérielle, c'est-à-dire qu'il identifie à quel gestionnaire de système décrit dans la configuration, il correspond. Il identifie également les autres *gestionnaires de systèmes* déployés sur l'infrastructure matérielle. Enfin, il identifie les opérations de déploiement qui le concernent, c'est-à-dire les *conteneurs* qu'il doit créer et paramétrer. L'ensemble de ces informations est traduit, dans la deuxième étape,

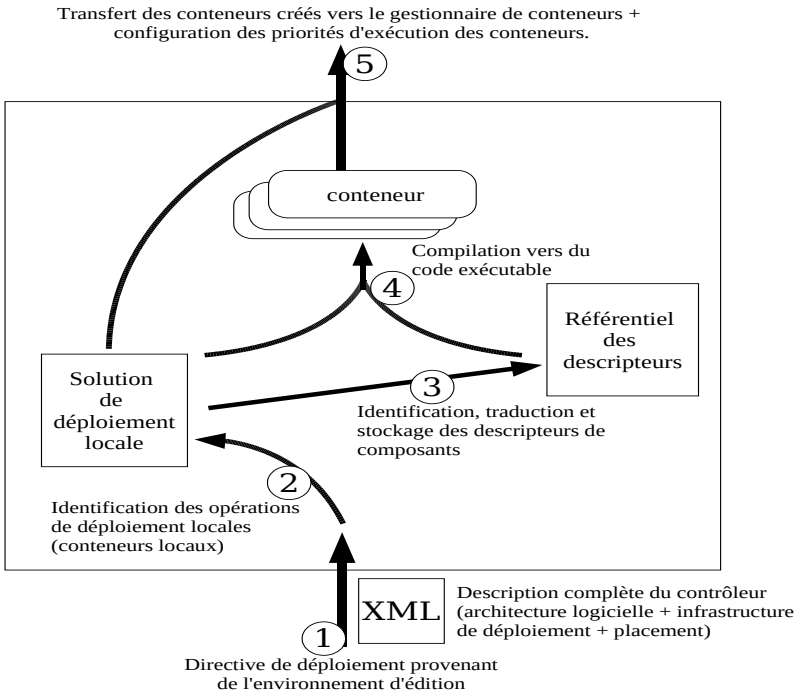


FIG. 8.2 – Schéma de la chaine de compilation d'un gestionnaire de déploiement

dans une solution de déploiement locale, enregistrée dans un fichier (XML). Cette solution locale contient toutes les opérations de déploiement que le gestionnaire de système doit réaliser sur le site.

- A partir de cette solution de déploiement locale, la troisième étape (3) consiste à identifier tous les descripteurs de composants utilisés au sein de tous les *conteneurs* déployés localement, à les transformer dans du code compilable (C++), et à les stocker dans un *référentiel des descripteurs*. Le *référentiel des descripteurs* est vu comme une table dont chaque entrée est associée à un fichier contenant le code source C++ d'un descripteur de composants.
- La quatrième étape (4) consiste à créer les *conteneurs*. Un *conteneur* (au sens code exécutable) est créé pour chaque *conteneur* (au sens information de description) identifié sur la solution de déploiement locale. La création d'un *conteneur* consiste à instancier les composants de contrôle et les composants rôles à partir de leurs descripteurs (présents dans le *référentiel des descripteurs*) et en fonction des informations contenues dans le *conteneur* correspondant de la solution de déploiement. Le fichier source (C++) de chaque *conteneur* est créé à cette étape, puis compilé sous la forme de code exécutable.
- La dernière étape (5) consiste à "transférer" les conteneurs ainsi compilés vers le *gestionnaire de conteneurs* qui sera configuré en fonction de leur information d'ordonnancement (priorités d'exécution système fixées en fonction des priorités relatives entre *conteneurs* établies par la solution de déploiement).

Dans un premier temps, le mécanisme du *gestionnaire de déploiement* n'est pas automatisé, mais il est réalisé de façon ad hoc. Les développeurs ont donc la charge de déployer le code compilé sur chaque site de l'infrastructure matérielle. Un mécanisme de déploiement automatisé permet d'envisager des opérations de reconfiguration à distance des architectures, sans que les développeurs aient besoin d'intervenir directement sur le matériel embarqué. Cette fonctionnalité nous semble, à terme,

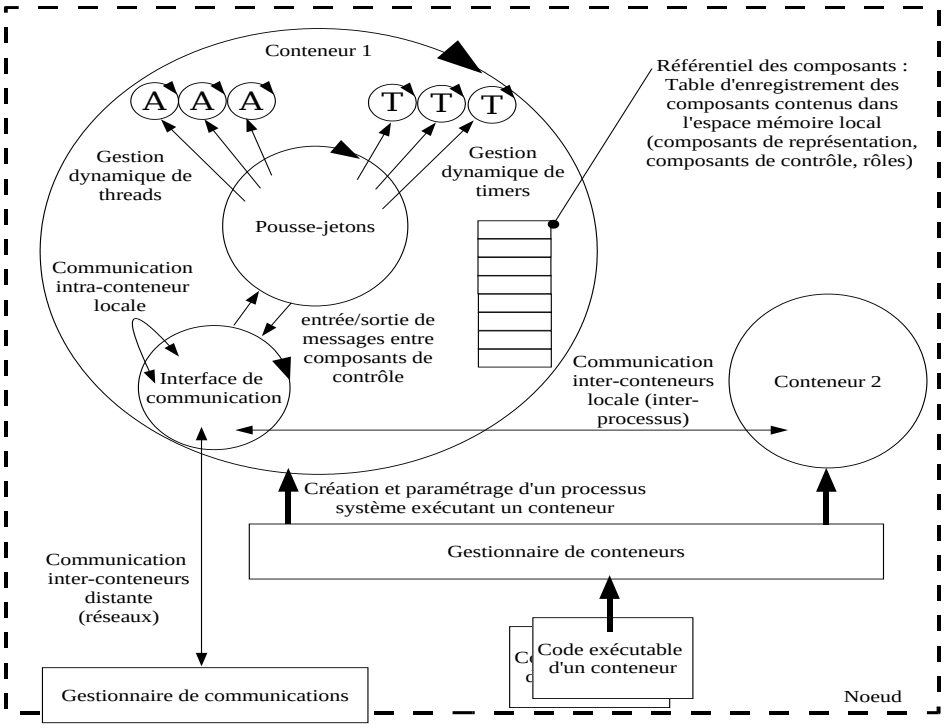


FIG. 8.3 – Schémas d'un gestionnaire de conteneurs et de conteneurs

essentielle à nombre de systèmes robotiques pour lesquels la distance avec l'opérateur ou l'environnement empêchent toute intervention humaine directe (e.g. la robotique d'exploration). De plus, le fait de déporter la compilation au niveau du site cible permet d'apdapter/optimiser la compilation en fonction de la nature de ce site (micro-contrôleur, micro-processeur, etc.) et/ou en fonction d'un système d'exploitation utilisé.

8.3 Conteneurs et Gestionnaire de conteneurs

Le *gestionnaire de conteneurs* est en charge de l'exécution des *conteneurs*. Il crée pour chaque *conteneur* le processus système sur lequel ce *conteneur* s'exécute. D'autre part, il paramètre l'ordonnancement des processus associés aux *conteneurs*, en fonction de l'information d'ordonnancement issue de l'étape précédente (cf. fig. 8.3).

La figure 8.3 est un schéma de l'exécution d'un *conteneur* par un *gestionnaire de conteneurs*. Chaque *conteneur* possède une table, nommée référentiel des composants, qui contient les références sur les composants existant dans son espace mémoire local (composants de représentation, composants de contrôle, composants rôles et indirectement composants connecteurs). Le référentiel des composants est nécessaire à toutes les opérations de recherche de composants au sein d'un conteneur. Un conteneur est constitué de deux entités s'exécutant en parallèle qui, chacune, peuvent accéder au *référentiel des composants* et qui communiquent via des envois de message asynchrones. Ces deux entités sont l'*interface de communication*, et le *pousse-jetons*.

Le *pousse-jetons* est l'entité en charge de l'exécution des comportements asynchrones des composants de contrôles et des composants rôles contenus dans un *conteneur*. C'est un exécuteur de RdPO, qui est capable de réaliser la propagation des jetons sur le graphe de contrôle d'un RdPO, en fonction

de la sémantique des RdPO. Il existe deux approches différentes pour implanter un contrôleur par le biais de réseaux de Petri [121]. La première consiste à traduire le réseau de Petri dans une collection de tâches (par le biais d'un langage procédural) de telle manière que cette collection émule la dynamique décrite par le réseau de Petri. La seconde consiste à considérer le réseau de Petri comme un ensemble de règles déclaratives et à exécuter cet ensemble de règles par le biais d'un mécanisme appelé pousse-jetons (plus connu sous le terme anglais de Token-Player). Nous avons choisi la deuxième solution, car elle permet aux développeurs d'éviter de transformer les réseaux de Petri sous forme procédurale. En effet le *pousse-jetons* ne dépend pas d'un réseau de Petri particulier, il se contente d'interpréter les règles représentant le réseau de Petri. Le *pousse-jetons* d'un conteneur est donc vu comme un exécuteur de réseaux de Petri à Objets, capable d'interpréter une structure informatique qui reflète exactement la structure et l'état du RdPO (état du graphe et état des données). Cette approche permet d'éviter une traduction d'un RdPO sous la forme de règles logiques à partir desquelles il serait difficile, à l'exécution, de reconstituer et de visualiser l'état et la structure du RdPO. En conséquence, on peut considérer la notation des RdPO comme un langage de programmation de haut niveau.

L'utilisation d'un tel *pousse-jetons* permet d'envisager différents mécanismes qui nous semblent essentiels pour le contexte applicatif ciblé :

- il est possible de retrouver, à tout moment de l'exécution, l'état de chaque composant de contrôle, ou composant rôle exécuté par un conteneur. Cet état correspond au vecteur de marquage du RdPO qui représente leur comportement asynchrone. Cette information peut par exemple être transmise à l'opérateur humain pendant une mission afin qu'il s'assure du bon fonctionnement d'un contrôleur. Pour cela un conteneur fournit des opérations de consultation du vecteur de marquage des comportements asynchrones (des composants de contrôle et des composants rôles) qu'il exécute.
- il est également possible de modifier le vecteur de marquage du RdPO d'un composant de contrôle ou composants rôle, via l'utilisation d'opérations de modifications fournies par un *conteneur*. Ceci permet d'envisager qu'un opérateur, un développeur ou un système de surveillance annexe, puisse remettre un contrôleur dans un état d'exécution particulier après, par exemple, qu'un dysfonctionnement ait été constaté.

Pendant son exécution, un *pousse-jetons* gère (création, destruction, paramétrage) un ensemble de *threads* (cf. fig.8.3), qui permettent de paralléliser certains calculs, ainsi qu'un ensemble de *timers*, qui permettent de générer des événements temporels. Le mécanisme d'exécution des RdPO, présenté plus en détail dans les sections suivantes, est, en partie, basé sur cette gestion des *threads* et des *timers*.

Le *pousse-jetons* communique avec l'*interface de communication* lorsque l'exécution du RdPO découle sur un envoi de messages, c'est-à-dire lorsqu'un jeton est placé dans une place de sortie. A l'inverse, l'*interface de communication* communique avec le *pousse-jetons*, dès lors qu'un des composants qu'il exécute reçoit un message, ce qui correspond au dépôt d'un jeton dans une place d'entrée. Une *interface de communication* est le biais par lequel sont routés les messages reçus et émis par le *pousse-jetons* (et donc par les composants exécutés). Elle est configurée en fonction des connexions asynchrones internes aux connecteurs (entre les ports asynchrones des composants rôles). Les échanges de messages peuvent être réalisés via des mécanismes différents en fonction du déploiement des connecteurs, c'est-à-dire en fonction du déploiement de leurs composants rôles. Les échanges sont sous-tendus par un des trois modes de communication possibles. Le premier mode de communication permet des échanges asynchrones de messages entre des composants rôles contenus dans le même conteneur (mode *intra-conteneur*, cf. fig. 8.3). Le second mode de communication permet des échanges asynchrones de messages entre des composants rôles contenus dans des conteneurs différents, mais ordonnancés par le même gestionnaire de conteneurs (mode *inter-conteneurs local*, cf. fig. 8.3). Enfin, le dernier mode de communication permet des échanges asynchrones de messages

entre des composants rôles contenus dans des conteneurs différents et ordonnancés sur des gestionnaires de conteneurs différents (mode *inter-conteneurs distants*, cf. fig. 8.3). Ce dernier mode est donc possible via l'établissement de communications réseaux entre gestionnaires de systèmes. L'*interface de communication* est dans ce cas en relation avec le *gestionnaire de communication*. L'utilisation d'un mode de communication est déduite directement du placement des connecteurs, c'est-à-dire de la répartition de leurs composants rôles dans un ou plusieurs conteneurs. Ainsi, le déploiement des composants de contrôle dans des conteneurs influe directement sur la façon dont les communications (induites par l'utilisation d'un connecteur) seront réalisées effectivement en terme de mécanismes système ou réseau.

Le *gestionnaire de conteneurs* et les *conteneurs* fournissent des services "techniques" pour gérer respectivement l'exécution des conteneurs et des composants. Le gestionnaire de conteneur permet de lancer/arrêter l'exécution d'un ou plusieurs *conteneurs* et de reparamétrer leurs propriétés d'ordonnancement. Un *conteneur* permet de réinitialiser les composants de contrôle et les composants rôles qu'il contient : cela revient à réinitialiser le marquage et arrêter tous les threads et timers associés au composant considéré. Il permet également de contrôler la propagation des jetons : bloquage/déblocage, notification de l'état du graphe du RdPO, application d'un nouveau marquage, verrouillage. Ces services "techniques" devraient, optimalement, être utilisés depuis l'environnement d'édition, afin de faciliter la travail des développeurs pendant la phase de test (ils pourraient également servir pendant la phase d'exploitation)

La compréhension du reste de ce chapitre nécessite de présenter en détail la notation utilisée (les réseaux de Petri à Objets) afin de comprendre le modèle de déploiement et d'exécution du langage.

8.4 Réseaux de Petri à Objets : formalisme et langage de programmation

Les mécanismes d'exécution du langage sont basés sur l'exécution des RdPO représentant les comportements asynchrones des composants de contrôle et des composants rôles. La notation formelle que nous utilisons est, en fait, une extension de celle des RdPO. Cette notation, que nous nommons RdPOC (pour réseau de Petri à Objets Communicant), est adaptée aux besoins du langage, et pour cela elle intègre les places d'entrée et les places de sortie.

La structure d'un RdPO est un graphe dont les nœuds sont des transitions et des places. Un arc est orienté et relie une transition à une place amont ou aval suivant l'orientation de l'arc. Un RdPO est constitué d'un ensemble de jetons distribués dans les places. Chaque jeton est un tuple d'objets dont la typologie (tuple de types) est compatible avec au moins une étiquette de la place qui le contient. A chaque place est associé un ensemble d'étiquettes qui définissent les typologies de jetons qu'elle peut contenir. Chaque arc possède un label listant un ensemble de variables, chaque variable ayant un type défini. Ce label définit la typologie des jetons filtrés par l'arc, c'est-à-dire les jetons que cet arc est capable de propager vers une place ou une transition. Chaque transition possède une partie *condition* qui liste les conditions que doivent satisfaire les objets contenus dans les jetons filtrés par les arcs amonts à cette transition. Chaque transition possède également une partie *action* qui détermine les actions réalisées sur les jetons, ou combinaisons de jetons, qui ont satisfait l'ensemble des conditions. Les actions modifient, créent ou détruisent des objets à partir desquels sont reconstitués les jetons en sortie de transition. Les jetons en sortie de transition sont créés en fonction des labels associés aux arcs avals à la transition. Formellement, nous définissons un RdPO marqué par le 5-uplet suivant :

$\mathbf{RPO} =< \mathbf{Pi}, \mathbf{T}, \mathbf{A}, \mathbf{Types}, \mathbf{V} >$, avec :

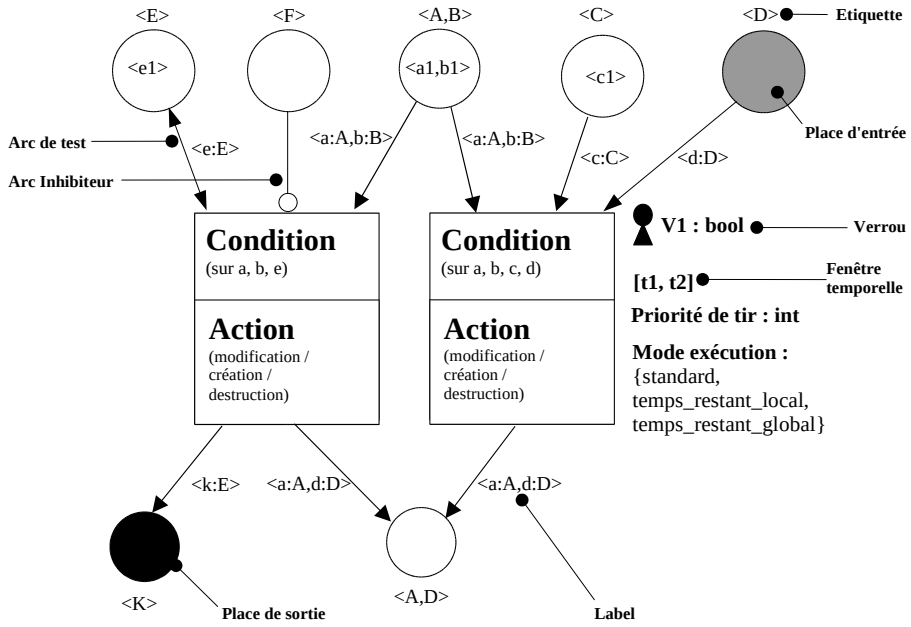


FIG. 8.4 – Notation RdPOC propre au langage CoSARC

- **Pi** est l'ensemble fini des places internes. Une place interne est un 2-uplet $\mathbf{Pla} =< \mathbf{Stp}, \mathbf{M_0} >$ tel que :
  - **Stp** est un ensemble fini d'étiquettes de la place. Une étiquette est un n-uplet contenant des éléments de **Types** : $\forall s \in \mathbf{Stp}, s \in \mathbf{Types}$.
  - $\mathbf{M_0}$ est le marquage initial de la place, constitué d'un ensemble de jetons. Un jeton est un n-uplet contenant des éléments instances des éléments de **Types**.
- **T** est l'ensemble fini des transitions. Une transition est un 2-uplet $\mathbf{Tra} =< \mathbf{Atc}, \mathbf{Ata} >$, avec :
  - **Atc** est une condition sur un ensemble de variables de **V**.
  - **Ata** est une action (operation) sur un ensemble de variables de **V**.
- **A** est l'ensemble fini des arcs. Un arc est un 3-uplet $\mathbf{Arc} =< \mathbf{Prec}, \mathbf{Foll}, \mathbf{Vars} >$, avec :
  - **Prec** représente le nœud précédent.
  - **Suiv** représente le nœud suivant.
  - **Vars** est le label qui décrit le filtre associé à l'arc. ce label est une somme formelle de n-uplets contenant des éléments de **V**. Par exemple `<x1,x2>` est un n-uplet et `<x1,x2>+<x3>` est une somme formelle de n-uplet.
- **Types** est l'ensemble fini des types de variables.
- **V** est l'ensemble fini des variables, chacune typée par un type de **Types**. Les variables sont identifiées de manière unique par leur nom et l'indentifant de leur RdPO.

Un RdPOC marqué est une extension qui rajoute à ce modèle les places d'entrée et de sortie. Formellement, un RdPOC est défini par le 7-uplet suivant :

$\mathbf{RPOC} =< \mathbf{Pi}, \mathbf{Pe}, \mathbf{Ps}, \mathbf{T}, \mathbf{A}, \mathbf{Types}, \mathbf{V} >$, avec :
- $\mathbf{Pi}, \mathbf{T}, \mathbf{A}, \mathbf{Types}, \mathbf{V}$ les mêmes ensembles que précédemment.
- **Pe** est l'ensemble fini des places d'entrée, avec :
  - $\mathbf{Pe} \cap \mathbf{Pi} = \emptyset$.
  - $\mathbf{Pe} \cap \mathbf{Ps} = \emptyset$.

  - $\forall p \in \mathbf{Pe}, \mathbf{M_0}(p) = \emptyset$.
  - $\forall p \in \mathbf{Pe}, \neg\exists a \in \mathbf{A}|\mathbf{Suiv}(a) = p$
- **Ps** est l'ensemble fini des places de sortie, avec :
  - $\mathbf{Ps} \cap \mathbf{Pi} = \emptyset$.
  - $\forall p \in \mathbf{Ps}, \mathbf{M_0}(p) = \emptyset$.
  - $\forall p \in \mathbf{Ps}, \neg\exists a \in \mathbf{A}|\mathbf{Prec}(a) = p$.

L'opération qui consiste à transformer un RdPOC en RdPO est notée $elim : RPOC \rightarrow RPO$. Elle consiste simplement à supprimer les places d'entrée et les places de sortie et les arcs qui les relient aux transitions. Une fois un RdPO marqué créé, on possède un modèle sur lequel peuvent être appliquées des méthodes d'analyse. Pour l'analyse, l'idée est de transformer le RdPO en réseau de Petri coloré (RdPC) équivalent au niveau comportemental [77], puis d'utiliser les nombreuses techniques d'analyse des RdPC [74]. L'opération *elim* est donc utilisée uniquement pour pouvoir obtenir un modèle analysable d'un comportement (d'un composant, d'une architecture ou d'un conteneur).

Formellement, étant donné un RdPOC *rpoc* et un RdPO résultat *rpo*, réaliser $elim : RPOC \rightarrow RPO$ correspond à appliquer les traitements suivants :

1. $\mathbf{Pi}(rpo) \leftarrow \mathbf{Pi}(rpoc)$.
2. $\mathbf{A}(rpo) \leftarrow \bigcup\{a \in \mathbf{A}(rpoc)|(\mathbf{Prec}(a)\neg \in \mathbf{Pe}(rpoc))et(\mathbf{Suiv}(a)\neg \in \mathbf{Ps}(rpoc))\}$.
3. $\mathbf{T}(rpo) \leftarrow \bigcup\{t \in \mathbf{T}(rpoc)|\exists a \in A|((\mathbf{Suiv}(a) = t)ou(\mathbf{Prec}(a) = t))\}$.
4. $\mathbf{V}(rpo) \leftarrow \bigcup\{v \in \mathbf{V}(rpoc)|(\exists a \in \mathbf{A}(rpo)|v \in \mathbf{Vars}(a))\}$.
5. $\mathbf{Types}(rpo) \leftarrow \bigcup\{typ \in \mathbf{Types}(rpoc)|\exists v \in \mathbf{V}(rpo)|(\mathbf{type}(v) = typ)\}$.

Notons que les actions (**Ata**) et les conditions (**Atc**) associées aux transitions ne sont pas utilisées pour l'analyse des RdPO. Nous avons donc simplifié la définition de *elim* sans prendre en compte l'impact de la suppression d'arcs sur les transitions : certains variables utilisées au sein des actions et des conditions n'existent plus après suppression des arcs.

Les places d'entrées (e.g. la place grisée d'étiquette `<D>`, fig.8.4) et les places de sortie (e.g. la place noire d'étiquette `<K>`, fig.8.4) ont été ajoutées pour permettre la formalisation des compositions et des communications des RdPOC et sont interprétées de manière spécifique par le *pousse-jetons*. Par ailleurs, la notation RdPOC a été enrichie d'un ensemble d'informations utiles à leur exécution (fig. 8.4) :

- Les *arcs de test* sont des arcs entrant de transitions, qui permettent de décrire un test sur la présence d'un jeton d'une typologie donnée dans une place donnée. Par exemple, l'arc labellé par `<e>` est un arc de test (arc représenté par une double flèche). La sémantique de l'arc de test, implique que le jeton filtré par cet arc n'est jamais modifié ni lu dans la transition, seule sa présence dans la place amont à la transition étant considérée. Si un jeton satisfaisant la typologie décrite par le label de l'arc de test est présent dans la place amont, la transition peut être tirée.

- Les *arcs inhibiteurs* (arc dont l'extrémité est un rond) sont des arcs amonts à des transitions, qui permettent de décrire un test sur l'absence de jetons dans une place donnée. La sémantique de l'arc inhibiteur implique qu'une transition ne pourra être tirée que si la place amont considérée ne contient aucun jeton. Au moment de l'édition d'un RdPOC, les arcs inhibiteurs et les arcs de test servent principalement à faciliter sa description. Au moment de l'analyse, le graphe doit être transformé en un graphe RdPO équivalent, de manière à ne contenir que des arcs standards. Leur sémantique est utilisée spécifiquement au moment de l'exécution afin d'optimiser les tests sur la présence de jetons.

- les *verrous* sont des informations associées aux transitions (e.g. `V1`, cf. fig. 8.4). Chaque transition peut posséder un verrou qui est défini à partir d'un ensemble de variable booléennes et

une même variable booléenne peut être utilisé dans plusieurs verrous. Un verrou permet de définir si la transition est tirable ou non. Si un verrou a pour valeur "vrai" (i.e. au moins une de ses variables booléenne a pour valeur "vrai"), l'ensemble des transitions verrouillées par celui-ci ne sont plus tirables. Elles le redeviennent dès lors que le verrou reprend pour valeur "faux" (i.e. toutes ss variables booléennes ont pour valeur "faux"). Les verrous permettent de contrôler différents chemins d'exécution possibles sur un RdPOC. Un conteneur permet à un utilisateur de verrouiller ou de deverrouiller certains chemins d'exécution, ce qui est très utile pendant des phases de test.

- Les *fenêtres temporelles* sont des informations associées à des transitions (e.g. `[t1, t2]`, cf. fig. 8.4)). Une fenêtre temporelle définit la date "au plus tôt" (`t1`) et la date "au plus tard" (`t2`) du tir d'une transition après sa sensibilisation (i.e. lorsque la distribution de jetons dans le graphe peut permettre le tir de la transition). Elle représente une contrainte sur la possibilité de tir de la transition à partir du moment où celle-ci est sensiblisée avec un ensemble de jetons répartis dans ses places amonts. La date au plus tôt représente donc un délai avant possibilité du tir et la date au plus tard représente le délai avant impossibilité définitive du tir. Les fenêtre temporelles sont utiliséees pour décrire, par exemple, des mécanismes de "chien de garde" ou des comportements périodiques.
- Les *priorités de tir* sont des informations associées aux transitions. Elles permettent de lever l'ambiguïté d'un RdPOC, ce qui est essentiel pour les exécuter. Une indéterminisme existe lorsque deux transitions sont en conflit potentiel, c'est-à-dire lorsqu'elles sont reliées à une même place amont avec des arcs filtrant des typologies de jetons compatibles (ce qui est le cas des deux transitions sur la figure 8.4). Dans le cas d'un conflit effectif (cf. plus loin), il faut faire un choix parmis les transitions qui peuvent être tirées en utilisant un même jeton (dans le cas où le jeton satisfait les conditions et que les fenêtres temporelles se chevauchent) : on parlera alors de conflit effectif. Les priorités de tir associées à chacune de ces deux transitions permettent de déterminer quelle est la transition qui sera tirée dans le cas d'un conflit effectif. Dès lors, la priorité représente une résolution déterministe de ces situations de conflit.
- un *mode d'exécution* est une information associée à chaque transition qui définit la manière dont l'action d'une transition s'exécute. Les actions réalisées au sein des transitions peuvent être "longues", en ce sens que leur exécution requiert un temps considéré comme important vis-à-vis de la dynamique du pousse-jetons. Dans ce cas, l'exécution de ces actions ne doit pas bloquer le mécanisme de propagation (i.e. ne doit pas affecter la réactivité du composant). Nous proposons pour cela, un mécanisme permettant de paralléliser les actions réalisées au sein des transitions, lorsque cela se révèle nécessaire. Le *mode d'exécution* d'une transition permet de définir quelles actions seront exécutées en parallèle et avec quel niveau de priorité. Si elle est *standard*, l'action de la transition ne sera pas parallélisée. Ce mode est utilisé pour les actions considérées comme peu coûteuses vis-à-vis de la dynamique de l'exécution du RdPOC par le pousse-jetons. Si le mode d'exécution est en *temps-restant local*, cela signifie que l'action sera exécutée en parallèle à une priorité moindre que celle du pousse-jetons du conteneur considéré, afin que celui-ci puisse continuer à propager sur d'autres chemins du RdPOC, sans être bloqué sur la réalisation d'une action. Elle s'exécutera néanmoins prioritairement à tous les autres conteneurs moins prioritaires. Ce mode est utilisé pour les actions considérées comme coûteuses vis-à-vis de la dynamique de l'exécution du RdPOC par le pousse-jeton. Enfin, si le mode d'exécution est en *temps-restant global*, cela signifie que l'action sera réalisée en parallèle avec la plus faible priorité de tout le système. Ce mode d'exécution permet d'éviter qu'une action considérée comme non "temps-réel" ne préempte l'exécution de tous les conteneurs moins prioritaires, dans un soucis évident de garder le contrôle de la réactivité du système. Notons que quel que soit le mode d'exécution choisi, l'exécution d'une action correspondant au tir d'une transition ne peut pas être arrêtée ou remise en cause, mais uniquement retardée.

8.5 Déploiement des composants : constructions des conteneurs

Avant de présenter la manière dont le pousse-jetons d'un conteneur exécute les RdPOC, il est nécessaire de présenter la façon dont les conteneurs sont créés par le *gestionnaire de déploiement*. Un conteneur est constitué des entités déjà présentées : le *pousse-jetons* et l'*interface de communication*. Ces deux entités sont configurées en fonction des composants contenus dans le conteneur. Le pousse-jetons est configuré par les comportement asynchrones des composants de contrôle et composants rôles à exécuter. L'interface de communication est configurée par les échanges de messages entre composants rôles. La construction des conteneurs se déroule en plusieurs étapes successives présentées par la suite.

8.5.1 Répartition des comportement rôles

A partir de la solution de déploiement locale, on possède l'information de répartition des composants de contrôle dans les conteneurs. Il faut alors déterminer les lieux d'exécution des composants rôles. Cela se réalise simplement, en ajoutant au conteneur tous les composants rôles correspondants aux rôles joués par chaque composant de contrôle qu'il contient (cf. algorithme 1).

Algorithm 1 Repartition des rôles

étant donné un conteneur *con*
for all composant de contrôle *c* contenu dans *con* **do**
  for all port de contrôle *p* de *c* connecté **do**
    soit p' le port du composant rôle *cr* connecté à *p*
    ajouter *cr* à *con*
  end for
end for

Une fois cette étape finie, on sait où s'exécutent les composants rôles et les composants de contrôle. L'étape suivante consiste à compiler tous les RdPOC afin de créer un RdPOC global pour le conteneur.

8.5.2 Construction des RdPOC des conteneurs

La construction du RdPOC global exécuté par le pousse-jetons d'un conteneur est réalisée via la fusion des comportements asynchrones des composants rôles et des composants de contrôle. La fusion des comportements asynchrones nécessite de formaliser le processus général de fusion des RdPOC. Définissons tout d'abord la première opération, notée $fusionplace$, qui à partir d'un ensemble de places d'entrée et d'un ensemble de places de sortie, crée une unique place interne. Formellement, étant donné PE l'ensemble des places d'entrée, PS l'ensemble des places de sortie et pi la place interne résultat, réaliser $fusionplace$ équivaut à appliquer les traitements suivants :

$fusionplace : set\{placeentree\} \otimes set\{placesortie\} \rightarrow placeinterne$

1. $\mathbf{Stp}(pi) \leftarrow \bigcup_i \{\mathbf{Stp}(PE_i)\} \cup \bigcup_j \{\mathbf{Stp}(PS_j)\}$
2. $\mathbf{M_0}(pi) \leftarrow \emptyset$

Le processus de fusion de RdPOC s'appuie sur l'opération $fusionplace$. Ce processus est représenté par l'opération $fusion$, qui prend en paramètre un ensemble de RdPOC et un ensemble d'appariements, et les fusionne en un unique RdPOC. Un appariement est un 2-uplet $\mathbf{Ap} =< \mathbf{EnsE}, \mathbf{EnsS} >$ avec $EnsE$ un ensemble de places d'entrée et $EnsS$ un ensemble de places de sorties. Il représente les places d'entrée et de sortie qui doivent être fusionnées. Formellement, étant donné un ensemble de RdPOC *ensrpoc* et un ensemble d'appariements *ensap* passés en paramètre et un RdPOC *resrpoc* résultat, réaliser l'opération $fusion$ revient à appliquer les traitement suivants :

$fusion : set\{RPOC\} \otimes set\{appariement\} \rightarrow RPOC$

1. $\mathbf{Pi}(resrpoc) \leftarrow \bigcup_i\{\mathbf{Pi}(ensrpoc_i)\} \cup \bigcup_j\{\mathbf{fusionplace}(\mathbf{EnsE}(ensap_j), \mathbf{EnsS}(ensap_j)\}$
2. $\mathbf{Pe}(resrpoc) \leftarrow \bigcup_i\{p \in \mathbf{Pe}(ensrpoc_i) | \forall ap \in ensap, (p\neg \in \mathbf{EnsE}(ap) et (p\neg \in \mathbf{EnsS}(ap)\}$
3. $\mathbf{Ps}(resrpoc) \leftarrow \bigcup_i\{p \in \mathbf{Ps}(ensrpoc_i) | \forall ap \in ensap, (p\neg \in \mathbf{EnsE}(ap)) et (p\neg \in \mathbf{EnsS}(ap))\}$
4. $\mathbf{T}(resrpoc) \leftarrow \bigcup_i\{\mathbf{T}(ensrpoc_i)\}$
5. $\mathbf{A}(resrpoc) \leftarrow \bigcup_i\{a \in \mathbf{A}(ensrpoc_i) | \forall ap \in ensap, (\mathbf{Prec}(a)\neg \in (\mathbf{EnsE}(ap) \cup \mathbf{EnsS}(ap)) et (\mathbf{Suiv}(a)\neg \in (\mathbf{EnsE}(ap) \cup \mathbf{EnsS}(ap))\} \cup \bigcup_j\{a \in \mathbf{A}(ensrpoc_i) | (\mathbf{Prec}(a) \leftarrow \mathbf{fusionplace}(\mathbf{EnsE}(ap), \mathbf{EnsS}(ap)), if \exists ap \in ensap | \mathbf{Prec}(a) \in (\mathbf{EnsE}(ap) \cup \mathbf{EnsS}(ap))) ou (\mathbf{Suiv}(a) \leftarrow \mathbf{fusion place}(\mathbf{EnsE}(ap), \mathbf{EnsS}(tap)), if \exists ap \in ensap | \mathbf{Suiv}(a) \in (\mathbf{EnsE}(ap) \cup \mathbf{EnsS}(ap)))\}$
6. $\mathbf{Types}(resrpoc) \leftarrow \bigcup_i\{\mathbf{Types}(ensrpoc_i)\}$
7. $\mathbf{V}(resrpoc) \leftarrow \bigcup_i\{\mathbf{V}(ensrpoc_i)\}$

L'étape de fusion des comportements asynchrones consiste à fusionner les RdPOC des composants de contrôle avec les RdPOC des composants rôles auxquels ils sont associés. Cela revient formellement à effectuer l'opération définie dans l'algorithme 2.

Algorithm 2 Fusion des comportements asynchrones

étant donné un conteneur *con*, un RPOC *finalrpoc* intialement vide et un ensemble d'appariements *ensap* initialement vide {on recherche tous les appariements à réaliser}
for all composant de contrôle *cc* contenu dans *con* **do**
  for all composant rôle *cr* dont le port p' est connecté au port p de *cc* **do**
    for all port asynchrone $pa1$ de p' **do**
      soit $pa2$ le port asynchrone de p connecté à $pa1$
      soient $pl1$ la place associée à $pas1$ et $pl2$ la place associée à $pa2$
      soit *ap* un nouvel appariement
      if $pl1$ est une place d'entrée **then**
        ajout de $pl1$ à $\mathbf{EnsE}(ap)$
        ajout de $pl2$ à $\mathbf{EnsS}(ap)$
      else
        ajout de $pl1$ à $\mathbf{EnsS}(ap)$
        ajout de $pl2$ à $\mathbf{EnsE}(ap)$
      end if
      ajout de *ap* à *ensap*
    end for
  end for
end for{Tous les appariements sont connus, on réalise la fusion}
soit *ensrpoc* l'ensemble contenant les RPOC de tous les composants de contrôle et de tous les composants rôles contenus dans *con*
$finalrpoc \leftarrow \mathbf{fusion}$(ensrpoc, *ensap*)

Nous remarquons qu'après le processus de fusion, le RdPOC obtenu n'est pas forcément connexe. C'est le cas lorsque plusieurs composants de contrôle sont placés à l'intérieur d'un conteneur. Cela ne fait néanmoins aucune différence pour le pousse-jetons du conteneur, qui exécute toutes ces composantes connexes en même temps. La figure 8.5 donne donne un schéma simple représentatif de ce mécanisme : un assemblage de deux composants est réalisé en utilisant un connecteur `RequestReplyConnection`. On voit que le composant `Comp1` est déployé dans le `conteneur 2`. `Comp 1` joue le rôle `Requester`, il est relié au composant rôle `RequesterRole`. On remarque que les comportements asynchrones de `Comp 1` et de `RequesterRole` ont été fusionnés : les places `P1` et `P3` ont donné naissance à une nouvelle place interne `P1-P3`.

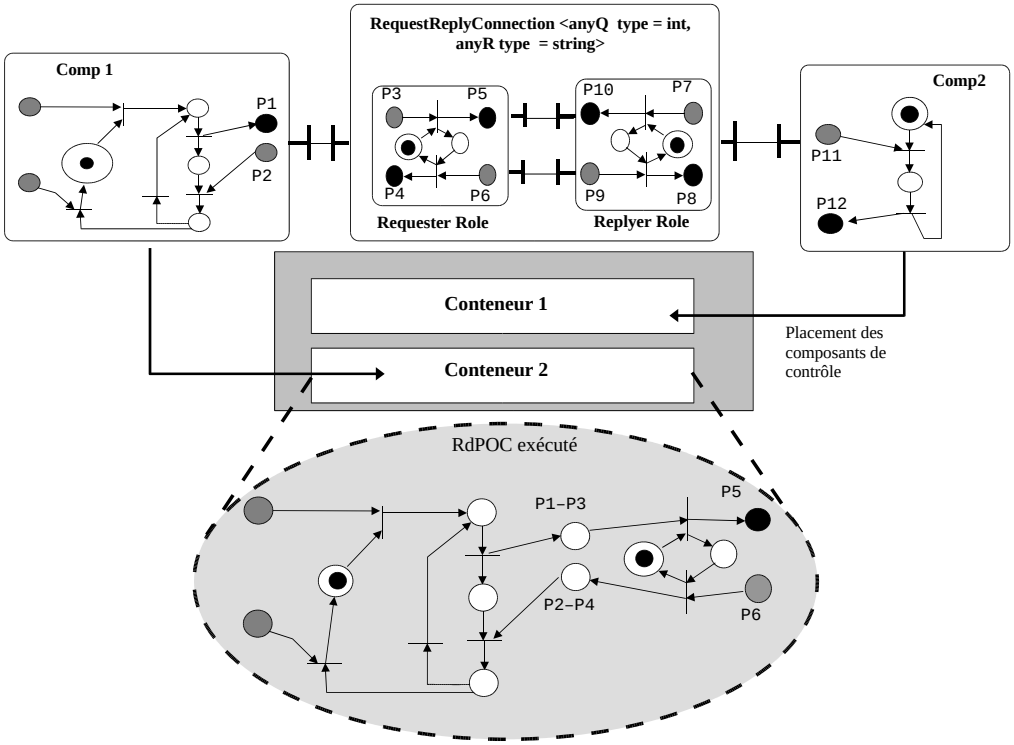


FIG. 8.5 – Construction des RdPOC des conteneurs

Dans le RdPOC global d'un conteneur, il reste encore des places d'entrée et de sortie. Ces places appartiennent soit aux comportements asynchrones des composants de contrôle, soit aux comportements asynchrones des composants rôles. Dans le premier cas, rien ne doit être fait puisque si des places d'entrée et de sortie des composants de contrôle existent encore, c'est la conséquence du fait que les ports de contrôle (fournis) correspondant n'ont pas été connectés. Les places d'entrée et de sortie des composants rôles sont, elles, utiles lors de l'étape de configuration des communications.

8.5.3 Décomposition des transitions

A partir du RdPOC global à un conteneur obtenu après réalisation de l'étape précédente, une étape de traduction de ce RdPOC est nécessaire. Cette traduction consiste à créer la structure informatique effectivement exécutée par le pousse-jetons. Cette structure informatique suit un modèle, que nous qualifions de *modèle RdPOC cible* qui est propre au mécanisme d'exécution implanté par le pousse-jetons. Un RdPOC cible conserve toutes les informations du modèle RdPOC initial et en enrichit certaines. La traduction du modèle initial vers le modèle cible a été spécifié dans [a1], dans le cadre d'une collaboration avec M. Combacau du LAAS.

Le modèle RdPOC cible est basé sur le principe de décomposition des transitions : chaque transition du RdPOC initial est décomposée en un graphe acyclique de *nœuds de transitions* (cf. fig 8.6). Les nœuds de transition sont de différents types :

- un *nœud de test* contient une opération de test s'appliquant aux objets contenus dans des jetons provenants d'une unique place amont. Cette opération de test fait partie des conditions associées à la transition équivalente dans le modèle RdPOC initial.
- un *nœud de jointure* contient une opération de test s'appliquant aux objets contenus dans des jetons qui proviennent de plusieurs places et/ou *noeuds de test* et/ou *noeuds de jointure*

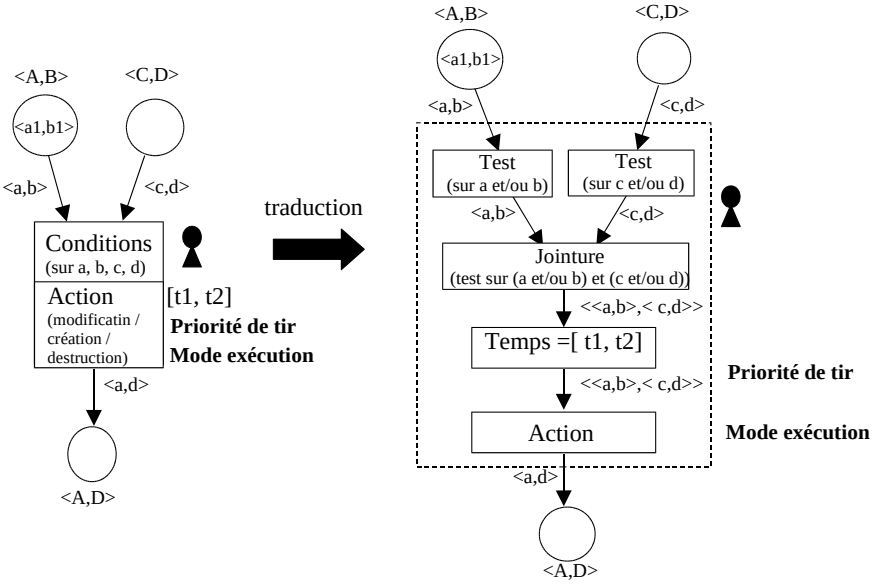


FIG. 8.6 – Exemple de décomposition de la structure d'une transition lors de la traduction d'un RdPO

amonts. Cette méthode de test fait partie des conditions associées à la transition équivalente dans le modèle RdPOC initial.

- un *nœud de temps* gère les événements de temps, liés à la spécification d'une fenêtre temporelle sur une transition. Il se base sur la "date au plus tôt" et sur la "date au plus tard", contenues dans la fenêtre temporelle, pour générer des *timers* qui produiront des événements lorsque la "date au plus tôt" et/ou la "date au plus tard" ont été atteintes.
- un *nœud d'action* contient les opérations associées à la partie action de la transition équivalente du modèle RdPOC initial. L'exécution de cette opération est réalisée lorsque le tir de la transition est effectif. Le nœud action est le seul obligatoirement présent dans la décomposition d'une transition (même si aucune action n'est associée à la transition).

Les nœuds de transition sont organisés de façon arborescente (la figure 8.6 donne un exemple typique de décomposition d'une transition). La structure arborescente est optimisée de manière à ne pas contenir de nœuds inutiles. Par exemple s'il n'y a pas de fenêtre temporelle sur la transition, celle-ci ne contiendra pas de nœud de temps. L'organisation de cette structure suit globalement les règles suivantes :

- Les nœuds de test sont rattachés aux places amont à la transition, lorsqu'une des conditions de la transition s'applique uniquement sur les jetons contenus dans cette place. Un nœud de test est rattaché en aval soit à un nœud de jointure (s'il y a plusieurs places amonts à la transition), soit à un nœud temps (si une fenêtre temporelle a été spécifiée pour la transition), soit directement à un nœud action (dans tous les autres cas).
- Les nœuds de jointure sont rattachés en amont à deux entités qui sont soit des places, des nœuds de test, des nœuds de jointure, soit encore une combinaison de ces possibilités. Un nœud de jointure n'est rattaché à une place amont que si un nœud de test n'est pas déjà rattaché à cette place. Sinon les nœuds de jointure sont rattachés à des nœuds de test ou à des nœuds de jointure. La partie condition d'une transition qui possède au moins deux places amonts, se décompose en un arbre de nœuds de transition, dont les feuilles sont des places amonts et/ou

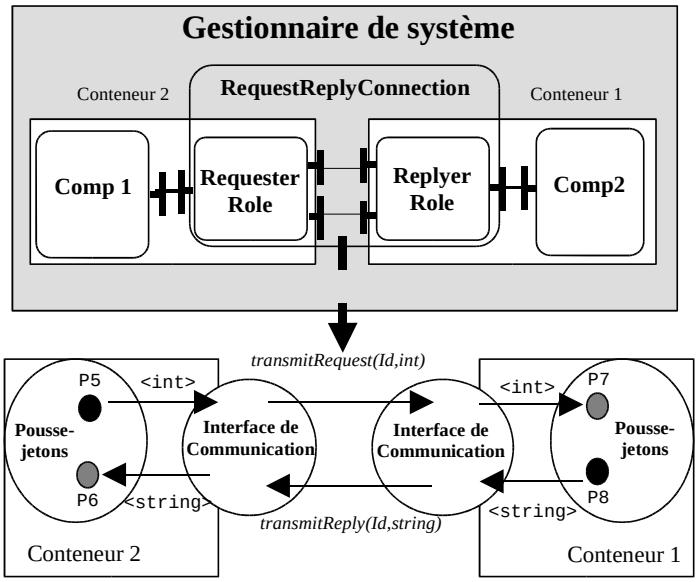


FIG. 8.7 – Configuration des communications et du routage des jetons échangés

des nœuds de test, les nœuds inermédiaires et la racine sont les nœuds de jointure. Par exemple, sur la figure 8.6, la partie condition de la transition se décompose en un arbre dont la racine est l'unique nœud de jointure et dont les feuilles sont les deux nœuds de test. Le nœud de jointure racine de l'arbre est celui auquel sont associés les arcs inhibiteurs et les arcs de test amonts à la transition.

- Le nœud de temps est automatiquement rattaché en amont soit à la racine de l'arbre de nœuds de jointure, soit à l'unique nœud de test (cas où il n'y a obligatoirement qu'une seule place amont). Il est obligatoirement rattaché en aval à un nœud d'action.
- Le nœud action est rattaché en amont, soit à un nœud de temps (cas où le noeud temps existe dans la décomposition), soit à un nœud de jointure (cas où il n'y a pas de nœud de temps et où la transition a plusieurs places amonts), soit à un nœud de test, soit à une place (cas où la partie condition de la transition est obligatoirement vide et où il n'y a qu'une seule place amont). En aval, le nœud action est relié à un ensemble de places dans lesquelles il dépose les jetons créés après le tir de la transition.

Notons que les autres informations associées aux transitions dans le modèle RdPOC initial (mode d'exécution, priorité, et verrous) sont toujours associées aux transitions dans le modèle RdPOC cible. Une fois que le RdPOC cible d'un conteneur a été compilé, son pousse-jetons est capable de l'exécuter.

8.5.4 Configuration de l'interface de communication

Les étapes précédentes ont permis de répartir et construire les RdPOC effectivement exécutés par chaque conteneur. Il reste donc à configurer les communications entre les conteneurs et au sein des conteneurs. Cette configuration est entièrement réalisée à partir des connexions asynchrones internes aux connecteurs, c'est-à-dire les connexions asynchrones entre ports asycnhrones des composants rôles (cf. fig. 8.7). C'est grâce aux connexions entre les composants rôles que les composants de contrôle pourront échanger des jetons suivant le protocole défini dans un connecteur.

La figure 8.7 reprend les composants et le déploiement présentés dans la figure 8.5. On voit que

chaque connexion asynchrone entre ports asynchrones des deux composants rôles, correspond à un échange de messages monodirectionnel (défini en fonction des interfaces typant les ports asynchrones connectés). Les *interfaces de communication* des conteneurs ont été configurées de manière à pouvoir router les jetons provenant des *pousse-jetons* sous la forme de message. Par exemple, à chaque fois que l'*interface de communication* du `conteneur 1` reçoit un jeton provenant de la place `P8`, il créé un message contenant les objets contenus dans le jeton, et émet ce message vers l'*interface de communication* du `conteneur 2`.

L'algorithme 3 décrit la procédure de configuration d'une *interface de communication*. On considère qu'une *interface de communication* possède une table de correspondance entre les messages reçu et les places d'entrée et une table de correspondance entre les messages émis et les places de sortie. Chacune de ces tables est constituée d'un ensemble de correspondances, chaque correspondance mettant en relation une place, un message et un point d'émission ou de réception de messages.

Algorithm 3 Configuration d'une interface de communication

étant donné un conteneur *con* et *ic* son interface de communication
soient *te* la table de correspondance des entrées et *ts* la table de correspondance des sorties de *ic*
for all composant rôle *cr* contenu dans *con* **do**
  for all port asynchrone p de *cr* **do**
    soit m le message décrit dans l'interface de p
    if p connecté à un port p' d'un conteneur con' **then**
      soit pl la place associée à p
      if pl est une place d'entrée **then**
        on créé un point de réception de messages prm
        on ajoute la correspondance $< pl, m, prm >$ à te
      else
        {pl est une place de sortie}
        on créé un point d'émission de messages pem
        on ajoute la correspondance $< pl, m, pem >$ à ts
      end if
    end if
  end for
end for

En fonction du point de réception de message dans lequel vient d'être déposé un message, l'*interface de communication* se charge de retrouver l'entrée correspondante dans la table de correspondance des entrées. A partir de cette entrée, elle possède les informations quant au format du message, elle reconstruit le jeton contenant les données associées à ce message. A partir de l'entrée de la table, elle connait également la place d'entrée destinataire, vers laquelle elle émet alors le jeton.

En fonction du jeton reçu en provenance d'une place de sortie, l'*interface de communication* se charge de retrouver l'entrée correspondante dans la table de correspondance des sorties. A partir de cette entrée, elle possède les informations quant au format du message, elle construit alors le message à partir du jeton. A partir de l'entrée de la table, elle connait également les destinataires possibles d'un message. Elle envoie simplement le message à travers l'unique point d'émission de message. Dans le cas où il y a plusieurs destinataires possibles, le point d'émission de messages se base sur la donnée `Id` contenue dans le message, pour router celui-ci vers le bon destinataire (sélection du receveur du message mais pas de diffusion du message).

La mise en œuvre des communciations entre *interfaces de communication* est fonction du déploiement des conteneurs. Dans le cas des communications intra-conteneur, aucun mécanisme système n'est utilisé, le routage des messages est local à l'*interface de communication*. Dans le cas des

communications inter-conteneurs locales, l'*interface de communication* s'appuit sur des primitives de communication système. Dans le cas des communications inter-conteneurs distantes, l'*interface de communication* s'appuit sur des primitives de communication réseaux fournies par le *gestionnaire de communication*.

Maintenant que la création des conteneurs a été expliquée dans les grandes lignes, nous expliquons dans la section suivante le mécanisme d'exécution des RdPOC implanté par son pousse-jetons.

8.6 Exécution des réseaux de Petri à Objets Communicants

Le mécanisme d'exécution des RdPOC du pousse-jetons a pour tâche de propager les jetons contenus dans un RdPOC en suivant les règles décrites par son graphe de contrôle, son interprétation (variables, conditions, actions), ainsi que les autres informations ajoutées à la notation. Une fois toutes les propagations possibles réalisées, le mécanisme se met dans un état dit stable, dans lequel il n'a plus d'activité et attend d'être "réactivé" à la suite de la réception de nouveaux évenements (événements temporels, messages, etc.).

8.6.1 Principe de propagation

Avant toute chose, nous présentons le principe de propagation qui s'appuit sur le modèle RdPOC cible. Le but, derrière l'utilisation du modèle RdPOC cible, est de limiter le nombre de tests : un test n'est réalisé qu'une seule fois sur chaque jeton ou combinaison de jetons. Si le test est réussi, le jeton (ou la combinaison de jetons) peut continuer sa propagation, s'il échoue il ne sera plus jamais propagé dans la transition. Cela permet au pousse-jetons d'être efficace, dans le sens où il limite au maximum le nombre de calculs (notamment le nombre de tests sur des combinaisons de jetons), et ce, même quand le RdPOC décrit un nombre important de synchronisations (impliquant potentiellement de nombreuses combinaisons) et/ou quand le marquage du graphe de contrôle est grand (i.e. quand il contient de nombreux jetons, potentiellement dans les mêmes places).

La figure 8.8 propose un exemple de propagation de jetons au sein de la transition de la figure 8.6. La propagation d'un jeton au sein d'une transition démarre à partir d'une place du RdPOC (étape 1, cf. fig. 8.8). Le jeton est filtré par l'arc amont à la transition en fonction de sa typologie (étape 2, le jeton `<a1,b1>` n'est filtré que par l'arc de label `<a,b>` et non par celui de label `<e,f>` dont la typologie n'est pas compatible avec celle du jeton) et il est propagé jusqu'au nœud de test. A ce niveau la méthode de test associée au nœud de test est exécutée (étape 3). Si le test est satisfait, le jeton continu sa propagation vers le nœud de jointure. Dans le cas contraire le jeton ne sera plus jamais propagé via cette transition. Au niveau du nœud de jointure, le jeton attend (étape 4) d'être combiné avec un jeton provenant de l'autre nœud de test : le mécanisme de propagation s'arrête. Lorsque qu'un jeton vient d'arriver dans la deuxième place (étape 5), le mécanisme de propagation reprend depuis cette place. Ce jeton est filtré par l'arc aval à cette place (étape 6) si sa typologie le permet. Il est propagé jusqu'au deuxième nœud de test (étape 7). A cette étape, la méthode de test associée au nœud de test est exécutée et le jeton continue sa propagation si le test est satisfait (dans le cas contraire il ne sera plus jamais propagé par la transition) et arrive au nœud de jointure. Là, une combinaison est créée à partir de ce jeton (`<c1,d1>`) et du jeton en attente depuis l'étape 4 (`<a1,b1>`). Le test associé au nœud de jointure est ensuite réalisé (étape 8) à partir des objets contenus dans cette combinaison (`<<a1,b1>,<c1,d1>>`). Si le test est satisfait, la combinaison de jetons poursuit sa propagation vers le nœud de temps, sinon la combinaison, ne sera plus jamais testée, elle est donc détruite. Au niveau du nœud de temps (étape 9), la combinaison de jetons est mise en attente : le nœud temps programme un *timer* (cf. fig 8.3) en fonction des informations sur la fenêtre de temps de possibilité de tir de la transition (`t1` correspondant au temps "au plus tôt",

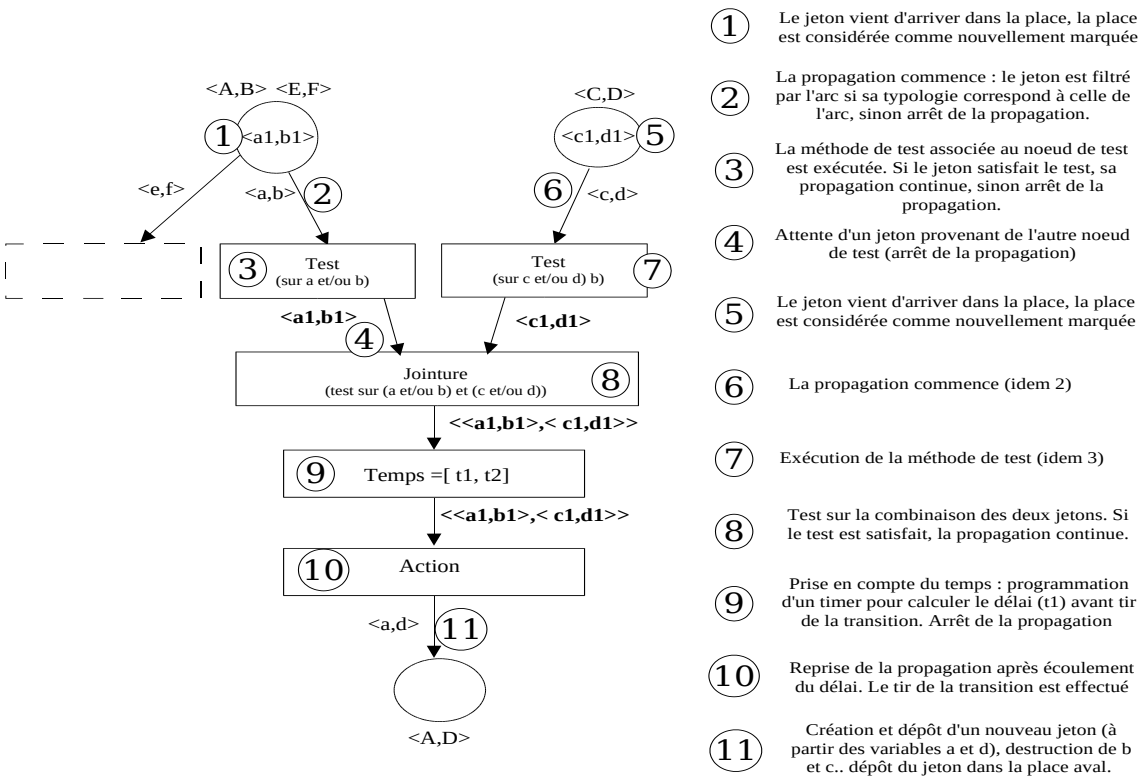


FIG. 8.8 – Exemple de propagation au sein d'une transition

et `t2` correspondant au temps de tir "au plus tard") et le mécanisme de propagation s'arrête. Sur réception d'une notification d'un événement de temps *fin-délai* (généré par le *timer* considéré une fois la période `t1` écoulée), la propagation est relancée, et la combinaison de jetons est propagée jusqu'au nœud d'action. Au niveau du nœud d'action (étape 10) l'opération associée est exécutée en fonction du mode d'exécution du nœud action (exécution parallèle ou non), avec pour paramètre les objets contenus dans la combinaison de jetons (la combinaison et les jetons eux-mêmes perdant leur existence). Dans le cas où l'action a été exécutée en parallèle, la propagation de la combinaison de jetons est alors arrêtée (mais les jetons restent indisponibles pour d'autres tirs). Lorsque le *thread* a terminé son exécution il émet un événement *fin-action* qui redémarre la propagation. A ce moment, des objets sont créés, détruits (cas de `b` et `c`) et/ou modifiés (e.g. `a`) : un nouveau jeton est reconstitué (dans l'exemple, il s'agit du jeton `<a1,d1>`, filtré par le label `a,d` de l'arc sortant de la transition) à partir de ces objets, et il est déposé dans la place aval à la transition (étape 11).

8.6.2 Procédures liées à la propagation au sein d'une transition

La mise en œuvre de ce principe de propagation est basé sur des procédures et des structures de données adaptées. La première procédure est chargée de propager les références de tuples aussi loin que possible dans une transition, en partant d'un noeud donné. La deuxième procédure est chargée de la résolution des conflits effectifs entre les transitions. La troisième procédure est chargée du retro-référencement des tuples au sein des transitions et places, une fois que les conflits effectifs ont été résolus.

Dans les procédures suivantes, on apelle *nœud*, indifféremment les places et les nœuds de transition et on apelle *tuple*, indifféremment les tuples d'objets (i.e. les jetons) et les tuples de jetons (i.e. les combinaisons de jetons). Ces trois procédures reposent sur les structures de données suivantes, dont une représentation graphique est donnée dans la figure 8.9 :

- chaque nœud contient une référence vers les nœuds précédents (plusieurs, uniquement dans le cas d'un nœud de jointure) et vers les nœuds suivants (plusieurs, dans le cas d'un nœud action ayant plusieurs places avals ou dans le cas d'une place ayant plusieurs transitions avals).
- une liste de propagation, notée `Lprop`, est associée à chaque nœud de transition. `Lprop` contient les références vers les tuples devant être propagés à partir du nœud considéré. Une place (place d'entrée ou place interne) contient une liste `Lprop` pour chacun de ses arcs avals (arc standard ou arc de test). Dans ce cas, la liste contient les références vers les tuples pouvant être propagés par l'arc standard considéré, ou vers les tuples dont la présence est testée par un arc de test.
- un ensemble de listes de test, notées `Ltest`, est associé à un nœud de jointure. Une liste de test stocke un ensemble de références sur des jetons (ou des combinaisons de jetons) entrants d'un nœud de jointure, qui proviennent d'un même nœud de transition amont ou d'une même place amont. Elle représente une mémoire des tuples en attente de combinaison avec d'autres tuples.
- une liste `Ltemps`, associée au nœud de temps d'une transition, stocke des références vers les tuples qui sont chacun en attente de réception d'un événement de temps pour reprendre leur propagation.
- un liste de marquage, notée `Lmarq`, est associée à chaque place. Elle contient les références vers tous les jetons que la place contient. Les arcs inhibiteurs se basent sur cette liste pour tester la présence de jetons dans une place.
- une liste des conflits détectés, notée `Lconflits` est globale à un RdPO. Elle contient des couples `(transition, set{tuple})` qui représentent des situations de conflits potentiels qui doivent être résolues en fonction des priorités de tir des transitions. L'ensemble des tuples associés à la transition dans ce couple, représente l'ensemble des tuples pouvant participer à un tir de cette transition.
- une liste `Latraiter` est associée au nœud action de chaque transition. Elle contient les tuples qui vont, chacun, participer à un tir de la transition considérée.

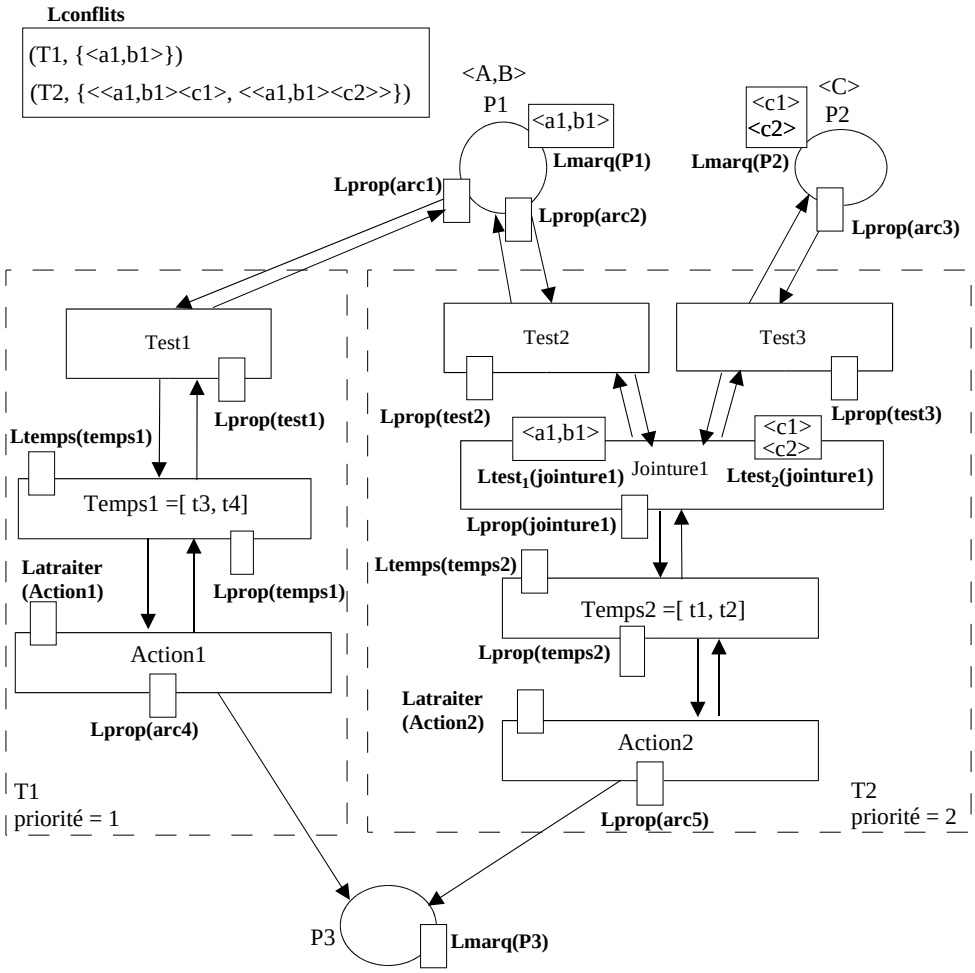


FIG. 8.9 – Structure informatique d'un RdPO

Les fonctions utilisées au sein des procédures sont spécifiées ci-après, mais nous ne les détaillons pas :

- La fonction `list lprop(noeudTransition n)` retourne la liste `Lprop` du nœud de transition `n`.
- La fonction `list lmarq(place p)` retourne la liste `Lmarq` associée à la place `p`.
- La fonction `list ltest(noeudjointure n, noeud prec)` retourne la liste de test associée au nœud de jointure `n`, liste qui correspond à la mémorisation des jetons provenant du nœud `prec`.
- La fonction `list ltemps(noeudtemps n)` retourne la liste `Ltemps` du nœud de temps `n`.
- La fonction `list lprop(place p, noeudTransition nt)` retourne la liste Lprop de la place `p` associée à l'arc aval à `p` et amont à `nt`.
- La fonction `list ltestcontient(tuple t, noeudjointure nj)` retourne la liste de test appartenant à `nj` qui contient le tuple `t`.
- La fonction `noeud suivant(noeudtransition n)` retourne le nœud suivant du noeud `n` dans la décomposition d'une transition.
- La fonction `noeudTransition suivant(place p, transition t)` retourne le nœud de transition aval à la place `p` et qui appartient à la transition `t`.
- La fonction `noeud précédent(noeudtransition n, list test = listvide)` retourne le nœud précédent du noeud `n` dans la décomposition d'une transition. Dans le cas où le nœud est un noeud de jointure il possède plusieurs prédécesseurs, l'information test permet de retrouver le prédécesseur du nœud de jointure rattaché à une liste de test spécifique de ce nœud.
- Les fonctions de manipulation des listes des nœuds qui permettent d'ajouter ou de retirer des tuples d'une liste. La fonction `void retire(list l, tuple t)` retire la référence vers le tuple `t`, de la liste `l`. La fonction `void add(list l, tuple t)` ajoute la référence vers le tuple `t`, de la liste `l`.
- La fonction `bool testtype(noeud n, NODETYPE T)` retourne "vrai" si le nœud `n` est de type `T`. Le `NODETYPE` est un type pouvant prendre pour valeur `TEST` (nœud de test), `JOINT` (nœud de jointure), `TIME` (nœud de temps) ou `ACTION` (nœud d'action).
- La fonction `bool test(noeud n, tuple t)` retourne "vrai" si le test associé au nœud `n` est vrai pour le tuple `t` et retourne "faux" dans le cas contraire.
- la fonction `bool verifSatisfaction(noeuddejointure n)` retourne "vrai" si la propagation à partir du nœud de jointure `n` considéré, n'est pas inhibée par des arcs de tests ou des arcs inhibiteurs. Dans le cas où le nœud de jointure n'est relié à aucun arc de test ou inhibiteur, la fonction retourne "vrai".
- La fonction `list combine(tuple t, Ltest lt, noeudjointure n)` retourne une liste des combinaisons créées entre le tuple `t` et les tuples contenus dans l'ensemble des listes de test du nœud `n`, à l'exception de la liste `Ltest` dans laquelle `t` seront ajoutées, après calcul, des combinaisons.
- La fonction `bool enConflit(transition T)` retourne "vrai" si la transition `T` est en conflit potentiel avec d'autres transition, et "faux" dans le cas contraire..
- La fonction `transition noeudAppartient(noeudTransition n)` est une fonction qui retourne une référence sur la transition à laquelle appartient un nœud de transition `n`.
- La fonction `noeudAction aPourAction(transition t)` retourne le nœud action d'une transition `t`.
- La fonction `inscrireConflit(transition tra, tuple tup)` permet d'ajouter le tuple `tup`, dans l'ensemble de tuples (i.e. le deuxième élément) du couple `(tra, set{tuples})`, lui-même contenu dans la liste `Lconflits`. Si la transition `tra` n'appartient à aucun couple, alors un nouveau couple est créé (i.e. le couple `(tra, set{tup})`, puis ajouté à la liste `Lconflit`.
- La fonction `void progTemps(transition T, tuple t)` permet de programmer le timer qui va se charger de générer les événements temporels induits par la fenêtre de temps d'une transition `T`, pour le tuple `t` considéré.

La procédure de propagation est présentée dans l'algorithme 4. Elle permet de propager des références sur des tuples aussi loin que possible au sein d'une transition (i.e. au sein de son graphe de décomposition en nœuds de transition). Elle prend en paramètre un nœud de départ de la propagation `dep` et un nœud vers lequel s'oriente la propagation `suiv`. Elle retourne une liste qui contient l'ensemble des jetons ayant atteint le nœud action d'une transition (c'est-à-dire l'ensemble des tuples prêts à participer au tir de la transition considérée). Cette procédure met en œuvre le principe de propagation décrit précédemment. Dans la figure 8.9, le jeton `<a1,b1>` a été propagé jusqu'au nœud de temps de la transition `T1`. Une fois le temps spécifié par la date de tir "au plus tôt" écoulé, le jeton a été propagé depuis la place `P1` jusqu'au nœud action de `T1` (on suppose donc qu'il a réussi le test de `Test1`). Néanmoins, puisque `T1` est en conflit potentiel avec `T2`, le jeton `<a1,b1>` n'a pas été placé dans la liste `Latraiter` du nœud `Action1`, mais a été enregistré dans la liste `Lconflit` (dans le couple `(T1,{<a1,b1>})`). La propagation reprend à partir de la place `P1` vers la transition `T2`. Le je ton `<a1,b1>` réussit le test de `Test2`, il est propagé jusqu'au nœud `Jointure1` où il est placé dans la liste de test `Ltest1(Jointure1)`. Au niveau du nœud de jointure `Jointure1`, il n'est pas combiné puisqu'on suppose que la liste `Ltest2(Jointure1)` est, à ce moment-là, vide. La propagation s'arrête et reprend, par la suite, à partir de la place `P2` à travers la transition `T2`. Les jetons `<c1>` et `<c2>` marquent la place `P2` et sont initialement présents dans la liste `Lprop(arc3)`. Ils sont propagés jusqu'au nœud de test `Test3` (ils sont supprimés de `Lprop(arc3)`), dont on suppose qu'ils réussissent le test, ils sont donc placés dans la liste `Lprop(Test3)`. Ils sont ensuite propagés vers le noeud `Jointure 1` (ils sont supprimés de la liste `Lprop(Test3)`). Là, ils sont chacun combinés avec les jetons contenus dans la liste de test `Ltest1(Jointure1)`, c'est-à-dire avec `<a1,b1>`. Deux combinaisons de jetons sont ainsi créées (`<<a1,b1><c1>>` et `<<a1,b1><c2>>`), et les jetons `<c1>` et `<c2>` sont placés dans la liste `Ltest2(Jointure1)`, en vue d'une éventuelle combinaison future avec de nouveaux jetons arrivant de `Test2`. On suppose que les deux combinaisons (i.e. tuples de jetons) créées réussissent le test de `Jointure1`, et sont placées dans la liste `Lprop(Jointure1)`. Elles sont ensuite propagées vers le nœud de temps (i.e. retirées de `Lprop(Jointure1)`) et sont placées dans la liste `Ltemps(Temps2)`. La propagation est arrêtée et reprend sur l'arrivée d'un événement stipulant que la date au plus tôt (i.e. `t1`) est atteinte (on suppose que d'autres propagations ont pu avoir lieu pendant ce temps) : les deux combinaisons reprennent leur propagation et elles sont placées dans la liste `Lprop(Temps2)` et retirées de la liste `Ltemps(Temps2)`. Leur propagation continue jusqu'au nœud `Action2`, mais comme `T2` est en conflit potentiel avec `T1`, ces deux combinaisons sont placées dans l'entrée de la liste `Lconflit`, correspondant à la transition `T2`. La figure 8.9 montre l'état des structures informatiques utilisées par la procédure de propagation. Les listes de propagation sont toutes vides. Par contre les listes de test et les listes `Lmarq` des places `P1` et `P2` contiennent encore des références aux jetons propagés.

La procédure de résolution des conflits effectifs apparaissant pendant l'exécution est primordiale pour notre modèle d'exécution. Elle assure le déterminisme de l'exécution du RdPOC, en fonction des priorités fixées sur les transitions. Avant toute chose, il convient de définir les différents types de conflits existants dans un RdPOC :

Un *conflit structurel* existe dès lors qu'au moins deux transitions sont reliées à au moins une même place amont. Formellement : $\forall t \in \mathbf{T}, t$ est en conflit structurel si et seulement si, $\exists a \in \mathbf{A} | (\mathbf{Suiv}(a) = t) et (\exists a' \in \mathbf{A} | \mathbf{Prec}(a) = \mathbf{Prec}(a'))$.

Une transition est en *conflit potentiel* si elle est en conflit structurel avec un ensemble de transitions et que l'interprétation (i.e. les labels associés aux arcs entrant des transitions, qui filtre les typologies de jetons propagés) rend possible la propagation d'un jeton dans plusieurs de ces transitions. Formellement : $\forall t \in \mathbf{T}, t$ est en conflit potentiel si et seulement si, t est en conflit structurel avec un ensemble de transitions $TRANS$ et $\exists t' \in \mathbf{TRANS} | (\exists a \in \mathbf{A} et \exists a' \in \mathbf{A} | (\mathbf{Suiv}(a) = t) et (\mathbf{Suiv}(a') = t') et (\mathbf{Prec}(a) = \mathbf{Prec}(a')) et (\forall i_{0 \rightarrow n}, Type(Vars_i(a)) = Type(Vars_i)(a')))$. Le conflit potentiel est

Algorithm 4 Propagation(noeud dep, noeudTransition suiv)

```
LIST ← listevide
if testtype(dep, PLACE) then
  ENS ← lprop(dep, suiv)
else
  ENS ← lprop(dep)
end if
for all référence sur un tuple t contenue dans ENS do
  if testtype(suiv, TEST) then
    retire(ENS, t)
    if test(suiv, t) then
      add(lprop(suiv), t)
      LIST ← Propagation(suiv, suivant(suiv))
    end if
  else if testtype(suiv, JOINT) then
    if verifSatisfaction(suiv) then
      retire(ENS, t)
      TEST ← ltest(suiv, dep)
      COMBIN ← combine(t, TEST, suiv)
      add(TEST, t)
      for all référence sur un tuple tup (correspondant à une combinaison de tuples) appartenant
      à COMBIN do
        if test(suiv, tup) then
          add(lprop(suiv), tup)
        end if
      end for
      LIST ← LIST + propagation(suiv, suivant(suiv))
    end if
  else if testtype(suiv, TIME) then
    retire(ENS, t)
    add(lprop(suiv), t)
    progtemps(noeudAppartient(suiv), t)
  else if testtype(suiv, ACTION) then
    retire(ENS, t)
    TRA ← noeudAppartient(suiv)
    if enConflit(TRA) then
      inscrireConflit(TRA, t)
    else
      add(LIST, t)
    end if
  end if
end for
return LIST
```

Algorithm 5 RésolutionConflits(list Lconflit)

$Lconflit \leftarrow sort(Lconflit)$ {sort() permet de trier les couples contenus dans la liste Lconflit, en fonction des priorités des transitions contenues dans les premiers éléments de ces couples}
for all couple C contenu dans la liste $Lconflit$ **do**
  $LTUP \leftarrow tuples(C)$ {tuples(C) renvoie la liste des références sur des tuples contenus dans le deuxième élément de C}
  $TRA \leftarrow trans(C)$ {trans(C) renvoie la transition contenue dans le premier élément de C}
  if $LTUP$ n'est pas vide **then**
    for all tuple t contenu dans $LTUP$, t est le vainqueur du conflit **do**
      for all jeton j appartenant à t **do**
        supprimer de $LTUP$ l'ensemble des tuples suivants contenant j
        for all couple C' contenu dans la liste $Lconflit$, suivant le couple C **do**
          $LTUP' \leftarrow tuples(C')$
          supprimer de $LTUPS'$ les tuples contenant j
        end for
      end for
    end for
  else
    C est retiré de la liste $Lconflits$, puis détruit.
  end if
end for
return $Lconflit$

détecté dès l'édition du graphe de contrôle du RdPO et chaque transition possède une variable qui détermine si la transition est en conflit potentiel ou non.

Un *conflit effectif* a lieu lorsque, au cours de la propagation d'un jeton J à travers au moins deux transitions T et T' en conflit potentiel, J satisfait toutes les conditions. Cela signifie que T et T' peuvent être tirées avec le même jeton, ce qui est interdit par la sémantique du formalisme. Un conflit effectif ne peut être découvert, et donc résolu, qu'à l'exécution.

La résolution de conflits effectifs est nécessaire après la propagation d'un ensemble de jetons à travers différentes transitions, lorsqu'un ou plusieurs jetons sont arrivés jusqu'au nœud action d'une transition, suite à l'appel de la procédure précédente. La procédure de résolution des conflits prend en paramètre une liste `Lconflits` et elle retourne cette liste modifiée ne contenant plus que les transitions à tirer, et pour chaque transition à tirer, l'ensemble des jetons concernés par un tir. Elle est présentée dans l'algorithme 5.

Une fois les conflits effectifs résolus, la liste `Lconflit` contient un couple pour chaque transition qui doit être tirée. Chaque couple contient l'ensemble des tuples qui vont amener à autant de tirs de la transition correspondante. Ces tuples sont ajoutés à la liste `Latraiter` du noeud action de la transition considérée. Un résultat de l'application de cette procédure à la liste `Lconflit` de la figure 8.9 est présenté dans la figure 8.10.

Après la résolution de conflit, l'état du graphe de nœuds de transition et des places du RdPOC n'est pas cohérent. En effet, la procédure de propagation propage des références sur des tuples le plus loin possible eu sein de toutes les transitions susceptibles de les laisser circuler, ce qui implique que ces références peuvent encore exister, dans des transitions en conflit potentiel avec la transition qui a gagné la résolution de conflit. De plus, des références à des jetons peuvent encore exister dans différentes listes de nœuds (e.g. listes `Ltest` des nœuds de jointure, liste `Ltemps` du nœud de temps) de la transition gagnante. Enfin, il existe encore des références à ces jetons dans les listes `Lprop` et

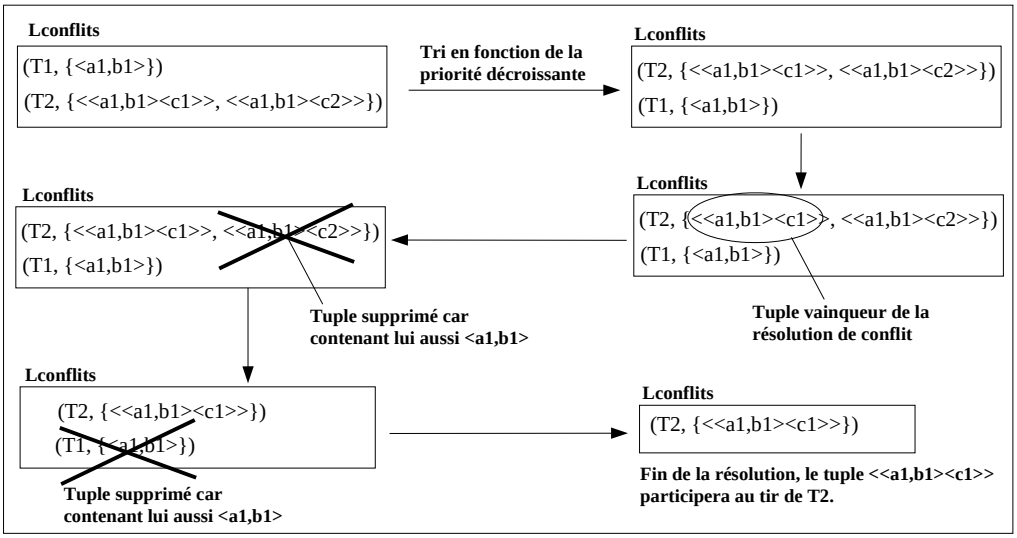


FIG. 8.10 – Exemple simple de résolution de conflits effectifs

Algorithm 6 Rétro-référencement-arrière(noeud courant, tuple tup, transition gagnante)

if $testtype(courant, ACTION)$ **then**
  return $Retro-referencement-arriere(precedent(courant), tup, gagnante)$
else if $testtype(courant, TIME)$ **then**
  $TEMPS \leftarrow ltemps(courant)$
  for all tuple t appartenant à $TEMPS$ **do**
    if t contient un des objets de tup **then**
      $retire(TEMPS, t)$
    end if
  end for
  $retire(lprop(courant), tup)$
  return $Retro-referencement-arriere(precedent(courant), tup, gagnante)$
else if $testtype(courant, JOINT)$ **then**
  for all tuple t appartenant à tup **do**
    $AVIDER \leftarrow ltestcontient(t, courant)$
    $retire(AVIDER, t)$
    $Retro-referencement-arriere(precedent(courant, AVIDER), t, gagnante)$
  end for
else if $testtype(courant, TEST)$ **then**
  return $Retro-referencement-arriere(precedent(courant), tup, gagnante)$
else
  {dans ce cas il s'agit d'une place et le tuple est un jeton}
  $retire(lmarq(courant), tup)$
  for all transition t aval à $courant$ en conflit potentiel avec $gagnante$ **do**
    $NODE \leftarrow suivant(courant, t)$
    $Retro-referencement-avant(NODE, tup)$
  end for
end if

`Lmarq` des places amonts de la transition. L'ensemble de ces références doit être supprimé après la résolution des conflits, car les jetons correspondants sont consommés lors du tir de la transition : ils ne doivent donc plus exister ailleurs que dans le nœud action de la transition considérée. La procédure de rétro-référencement est là pour remettre les structures de données du RdPOC dans un état cohérent, une fois la résolution de conflit effectuée et donc, implicitement, une fois les tirs des transitions décidés. Cette procédure est divisée en deux sous-procédures. La première, nommée `Rétro-référencement-arrière` est chargée du rétro-référencement des jetons en "remontant" à partir du nœud action d'une transition, jusqu'aux places amonts de la transition. Elle supprime toutes les références aux jetons participant aux tirs qui ont été décidés après résolution de conflits.

Algorithm 7 Rétro-référencement-avant(noeud courant, tuple tup)

if $testtype(courant, TIME)$ **then**
  $TEMPS \leftarrow ltemps(courant)$
  for all tuple t appartenant à $TEMPS$ **do**
    if t contient un objet de tup **then**
      $retire(TEMPS, t)$
    end if
  end for
else if $testtype(courant, JOINT)$ **then**
  $TEST \leftarrow ltestcontient(tup, courant)$
  for all tuple t appartenant à TEST **do**
    if t contient tup **then**
      $retire(TEST, t)$
    end if
    $Retro - referencement - avant(suivant(courant), t)$
  end for
else if $testtype(courant, TEST)$ **then**
  $Retro - referencement - avant(suivant(courant), tup)$
end if

La seconde procédure, nommée `Rétro-référencement-avant`, est chargée de supprimer les références à un jeton, depuis une place vers les transitions avals à cette place, si ces transitions sont en conflit potentiel avec la transition gagnante de la résolution des conflits. Elle supprime les références aux jetons en descendant à travers l'arbre de nœuds des transitions, jusqu'aux nœuds actions de ces transitions. Si l'on s'en réfère aux exemples précédents (cf. fig8.9), le rétro-référencement des jetons démarre à partir du nœud `Action2`, avec le tuple `<<a1,b1><c1>>` (cf. fig. 8.10) (qui se trouve désormais référencé dans la liste `Latraiter(Action2)`). Le rétro-référencement-arrière s'applique aux noeuds de transition en commençant par `Temps2` (rien n'est fait puisque `Ltemps(Temps2)` est vide). Il s'applique ensuite à `Jointure1`. Là, le tuple `<<a1,b1><c1>>` est divisé en deux tuples (jetons dans ce cas) : `<a1,b1>` et `<c1>`. La procédure identifie les listes de test contenant ces deux jetons (`Ltest1(Jointure1)` pour `<a1,b1>`, `Ltest2(Jointure1)` pour `<c1>`), et supprime les références à ces jetons dans les listes de test respectives. `Ltest1(Jointure1)` se retrouve alors vide et `Ltest2(Jointure1)` ne contient plus que le jeton `<c2>` qui pourra continuer à participer à des combinaisons. Le rétro-référencement-arrière s'applique alors récursivement aux deux nœuds de test amonts : `Test2` pour le jeton `<a1,b1>` et `Test3` pour `<c1>`. Dans les deux cas, rien n'est fait au niveau des nœuds de test et le rétro-référencement-arrière s'applique alors aux deux places amonts de `T2`. La référence à `<a1,b1>` est supprimée de `Lmarq(P1)` (qui se trouve alors vide) et la référence à `<c1>` est supprimée de `Lmarq(P2)` (qui ne contient plus que la référence à `<c2>`). La procédure de rétro-référencement-avant est alors lancée à partir de `P1` vers la transition `T1`. Dans l'exemple

présenté, le rétro-référencement-avant atteint le nœud action sans supprimer de référence, puisqu'il n'en existe plus dans `T1`.

8.6.3 Procédure d'exécution des RdPOC

La procédure globale d'exécution d'un RdPOC est présentée dans la figure 8.11. Elle se base sur les trois procédures et les fonctions présentées précédemment, ainsi que sur les structures de données et les procédures suivantes :

- une liste nommée `Pnm`, la liste des places nouvellement marquées. Une place nouvellement marquée est une place qui contient des jetons qui n'ont pas encore été propagés vers les transitions aval à la place. Cette liste permet de connaître les places qui viennent de recevoir des jetons, et donc à partir desquelles peuvent commencer des propagations de jetons, qui sont alors appelés jetons nouvellement arrivants.
- une liste nommée `Asna` qui contient les arcs spéciaux (arcs de test et arcs inhibiteurs) nouvellement actifs, c'est-à-dire qui peuvent potentiellement rendre possible le tir d'une transition.
- une liste nommée `Tnd` qui contient la liste des transitions nouvellement déverrouillées, c'est-à-dire les transitions qui peuvent à nouveau être suceptibles d'un tir.
- la procédure `répartition(place p, jeton j)` permet de répartir le jeton nouvellement arrivant `j` dans la place `p`. `j` est alors placé dans la liste `Lmarq(p)` puis il est distribué dans l'ensemble des listes `Lprop(arci)`, où `arci` est un arc aval de `p`, en fonction de sa typologie qui doit être compatible avec celle de `arci`.
- la procédure `émission(placesortie P)` permet d'émettre tous les jetons contenus dans la liste `Lmarq` d'une place de sortie `P` vers l'*interface de communication*.
- la procédure `préAction(noeudAction na, list tups)` permet de mettre la structure informatique à jour une fois les tirs de transitions décidés. Elle démarre d'un nœud action `na` et supprime tout d'abord de `tups` les tuples contenant des références redondantes aux mêmes jetons (i.e. résolution de conflits locaux à une transition pour éviter qu'une transition ne soit tirée plusieurs fois avec un même jeton). Elle appelle ensuite la procédure de rétro-référencement-arrière, afin de supprimer les références sur les jetons contenus dans les tuples de la liste `tups`. Ces tuples contiennent les objets qui vont participer au tir de la transition contenant `na`. `PréAction(noeudAction na, list tups)` met également à jour la liste `Asna`, en fonction des modifications du marquage des places amonts des transitions qui vont être tirées (suppression des arcs de test ayant été inactivés et ajout des arcs inhibiteurs ayant été activés).
- la procédure `list exécuter(noeudAction na, tuple tup)` permet d'exécuter la fonction du nœud action `na` avec pour paramètres les objets contenus dans le tuple `tup`. Elle retourne la liste des objets créés ou modifiés par la fonction.
- La fonction `estParallèle(transition t)` retourne "vrai" si l'exécution de l'action de la transition `t` est réalisée en parallèle, et "faux" dans le cas contraire.
- la procédure `paralléliser(noeudAction na, tuple tup)` permet de paralléliser l'exécution de la fonction `list exécuter(noeudAction na, tuple tup)`. La parallélisation est faite en fonction du mode d'exécution spécifié pour la transition associée au nœud `na`. Une fois l'exécution parallèle terminée, un événement de *terminaison-action* est envoyé au pousse-jetons : cet événement contient les objets créés ou modifiés par l'exécution de la fonction associée au nœud action.
- la procédure `créerJetons(noeudAction na, list objets)` permet, après le tir d'une transition, de créer les jetons en sortie du nœud action `na` à partir de la liste `objets`. Ces jetons iront remplir les listes `Lprop` associées aux arcs avals du nœud `na` (i.e. les arc reliant `na` à des places avals).
- La procédure `postAction(noeudAction na)` permet de propager les références aux jetons créés en sortie de transition (contenus dans les listes `Lprop` du nœud `na`), dans les places avals

du nœud action `na`. Cette procédure est appelée chaque fois qu'un tir d'une transition est terminé. Elle utilise, pour chaque place aval dans laquelle elle dépose des jetons, la fonction `repartition(place p, jeton j)` et ajoute ces places à la liste `Pnm`. Enfin, elle met à jour, en conséquence, la liste `Asna` (suppression des arcs inhibiteurs ayant été inactivés et ajout des arcs de test ayant été activés).

- La procédure `gérerEvénementExterne(message m)` permet de reconstruire le jeton j contenu dans le message `m` puis de placer j dans la place d'entrée p référencée par le message `m`. p est ajoutée à la liste `Pnm`, puis la fonction `repartition` est appelée avec pour paramètre p et j. En conséquence, elle met à jour, si besoin est, la liste `Asna` (suppression des arcs inhibiteurs ayant été inactivés et ajout des arcs de test ayant été activés).
- La procédure `gérerEvénementTemps(message m)` permet de continuer la propagation à partir du nœud de temps dont le délai, associé à la date "au plus tôt" de la fenêtre temporelle, est écoulé. Elle extrait du message `m` le nœud de temps nt concerné par l'événement de temps, ainsi que le tuple tup concerné. Si tup est encore contenu dans la liste `Ltemps` de nt, et si la transition associée n'est pas verrouillée, elle relance la propagation à partir de ce nœud. Si la propagation amène le jeton jusqu'au nœud action et que la transition considérée n'est pas en conflit potentiel, elle appelle `PréAction` et exécute ensuite la fonction associée au nœud action concerné (parallélisée ou non). Dans le cas où l'action n'est pas parallélisée, elle exécute sucessivement `créerJetons` puis `postAction` (finition immédiate du tir). Notons qu'à l'heure actuelle, le pousse-jetons n'intègre pas la gestion de la date de tir "au plus tard" de la fenêtre temporelle, mais il est envisagé de gérer celle-ci en programmant un deuxième timer qui signalerait lorsque la transition n'est plus tirable avec un tuple.
- La procédure `gérerEvénementAction(message m)` permet de gérer la terminaison d'une exécution parallèle d'une fonction. Elle extrait du message `m` le nœud action considéré na ainsi que la liste lo des objets retournés. Elle appelle ensuite successivement la fonction `créerJetons`, avec pour paramètres na et lo et la fonction `postAction` avec pour paramètre na.
- La procédure `gérerArcsNouvellementActifs(list Asna)` permet de gérer la conséquence de l'activation d'arcs spéciaux. Pour chaque arc spécial as nouvellement actif, contenu dans la liste `Asna`, elle retire as de `Asna` et reprend la propagation à partir du nœud de jointure nj associé à as (si d'autres arcs spéciaux reliés à nj ne sont pas inactifs et si la transition associée à nj n'est pas verrouillée). Lorsqu'un nœud action a été atteint, elle exécute `PréAction` puis lance l'exécution de la fonction associée (i.e. appel de `exécuter` ou `paralléliser`). Dans le cas où l'action n'a pas été parallélisée, elle exécute sucessivement `créerJetons` puis `postAction` (finition immédiate du tir).
- La procédure `gérerTransitionsNouvellementDéverrouillées(list Tnd)` permet de gérer la conséquence du déverrouillage de transitions. Pour chaque transition t nouvellement déverrouillée, la propagation est relancée à partir de chacune des places amonts de t, vers t. Quant un nœud action a été atteint, elle exécute `PréAction` puis lance l'exécution de la fonction associée (i.e. appel de `exécuter` ou `paralléliser`). Dans le cas où l'action n'a pas été parallélisée, elle exécute sucessivement `créerJetons` puis `postAction` (finition immédiate du tir).
- La procédure `gérerRésolutionConflits(list Lconflit))` résoud les conflits effectifs (appel de `résoudreConflit` avec `Lconflit` en paramètre). Elle appelle la procédure `PréAction` (rétro-référencement) à partir des nœuds actions des transitions devant être tirées, et ceci pour chaque tuple vainqueur contenu dans la liste `Lconflit`. Pour chacun de ces tuples, elle lance l'exécution de la fonction associée (i.e. appel de `exécuter` ou `paralléliser`). Dans le cas où l'action n'a pas été parallélisée, elle exécute sucessivement `créerJetons` puis `postAction` (finition immédiate du tir).
- La procédure `gérerNouvellePropagation(place P, noeudTransition nt)` permet de propager les jetons nouvellement arrivants contenus dans la place `P`, vers le nœud de transition nt. Cette procédure est présentée dans l'algorithme 8. Les algorithmes des procédures de ges-

tion des événements, des conflits, des arcs nouvellement actifs et des transitions nouvellement déverrouillées (cf. précédemment) sont des déclinaisons de cet algorithme.

Algorithm 8 gérerNouvellePropagation(place dep, noeudTransition suivant)

```
if noeudAppartient(suivant) n'est pas verrouillée then
  LIST ← Propagation(dep, suivant)
  if LIST n'est pas vide {un jeton au moins a atteint le noeud action} then
    TRANS ← noeudAppartient(suivant)
    ACT ← aPourAction(TRANS)
    PreAction(ACT, LIST) {retro-référencement et mise à jour Asna}
    for all tuple t appartenant à LIST do
      if estParrallele(TRANS) then
        paralleliser(ACT, t) {l'action est exécutée en parallèle et sa priorité est fixée en fonction
        de son mode d'exécution}
      else
        OBJS ← executer(ACT, t)
        creerJetons(ACT, OBS)
        postAction(ACT)
      end if
    end for
  end if
end if
```

La procédure globale d'exécution des RdPOC (cf. fig. 8.11) est basée sur la recherche d'un *état stable*. Le pousse-jetons atteint un *état stable* lorsque plus aucune propagation n'est possible dans la structure de contrôle du RdPOC exécuté. Il est alors en attente d'un événement déclencheur qui vienne le sortir de cet état stable. Ces événements déclencheurs peuvent être des événements internes, c'est-à-dire des événements de "temps écoulé" ou des événements de "terminaison d'action parallèle", générés respectivement par des *timers* et des *threads* qu'il a programmé. Ces événements internes vont permettre de réactiver la propagation au sein des transitions : la poursuite de la propagation d'un jeton vers le nœud action d'une transition (cas d'un événement de temps) ou le dépôt de jetons dans les places avals à la transition, places qui seront alors considérées comme nouvellement marquées (cas d'un événement de terminaison d'action). Les événements déclencheurs peuvent aussi être externes, c'est-à-dire que ces événements correspondent à des messages entrants. Ces messages entranst sont traduits en un dépôt d'un jeton dans une place d'entrée du RdPOC. Lorsque le pousse-jetons reçoit des événements déclencheurs, il en déduit alors les propagations à réaliser et sort de son *état stable*. Notons que l'état stable ne signifie pas nécéssairement que le pousse-jetons n'est pas actif, car des *threads* qu'il a généré peuvent s'exécuter en parallèle.

Cette procédure d'exécution globale se base sur la liste des places nouvellement marquées `Pnm`, qui initialement, contient l'ensemble des places initialement marquées. Elle prend successivement chaque place (interne) nouvellement marquée de `Pnm` et propage tous les jetons nouvellement arrivants dans cette place `Pi`, aussi loin que possible dans chacune des transitions avals de `Pi` (appel à `gérerNouvellePropagation`). Si des nœuds de temps sont atteints, cela peut donc déboucher sur la programmation de *timers* chargés de générer des événements de temps et dans ce cas cela arrête la propagation. Si la propagation des références se fait jusqu'au nœud action et que la transition est en conflit potentiel, la propagation s'arrête et reprendra après la résolution de conflits (si la transition est gagnante). S'il n'y a pas de conflit potentiel, la référence au jeton est supprimée par rétro-référencement (afin d'éviter qu'un jeton ne puisse être impliqué dans plusieurs tirs de la même transition). Si le nœud action programme un ou plusieurs *threads d'action*, la propagation des je-

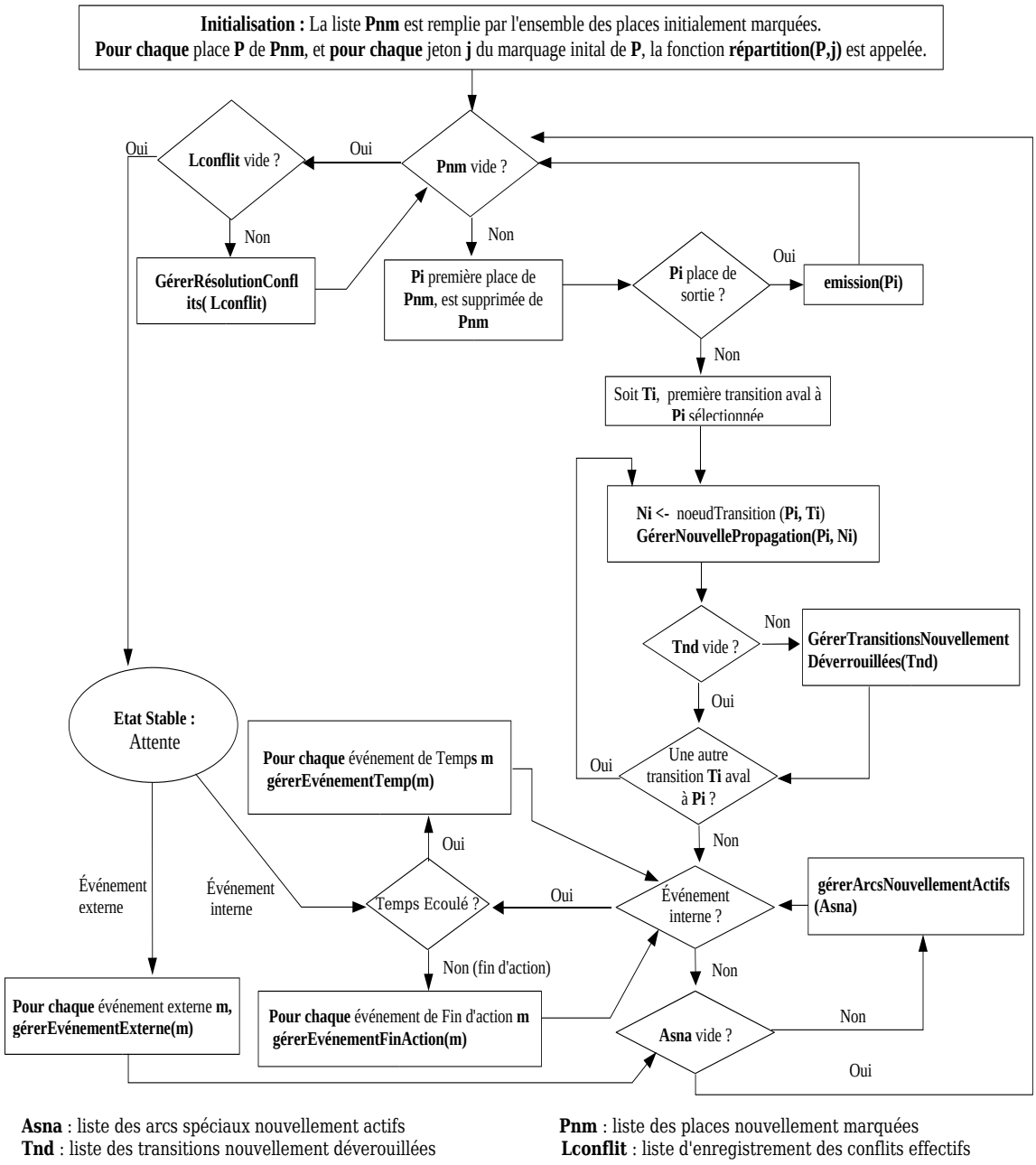


FIG. 8.11 – Procédure d'exécution d'un RdPOC (simplifiée)

tons concernés s'arrête et reprendra lorsque le *thread* enverra un événement de terminaison d'action. Si, après la propagation, des nouveaux jetons ont été créés et déposés dans des places, ces places sont alors considérées comme nouvellement marquées et placées dans `Pnm`. La procédure d'exécution continue ainsi pour chaque place jusqu'à ce qu'il n'y est plus de place nouvellement marquée dans la liste `Pnm`. Entre chaque tir de transition pour une même place amont, la procédure lit la liste `Tnd` (transitions nouvellement déverrouillées) afin de réaliser d'éventuelles propagations liées au déverrouillage de transitions. Entre chaque propagation à partir d'une place de `Pnm`, la procédure regarde si de nouveaux événements internes ont été reçus, et réalise les propagations adéquates si c'est le cas (appel de `gérerEvénementFinAction` ou `gérerEvénementTemps`). Entre chaque propagation à partir d'une place, la procédure regarde aussi si de nouveaux arcs spéciaux ne sont pas nouvellement actifs et, si c'est le cas, réalise les propagations adéquates.

Lorsque plus aucune propagation n'est possible à partir d'une place, la procédure résoud les éventuels conflits existant entre des transitions prêtes à être tirées. Une fois ceci fait, elle réalise les tirs correspondants ainsi que (si ces tirs ne sont pas parallélisés) la propagation vers les places avals des transitions concernées. Lorsque les listes `Pnm` et `Lconflits` sont vides, le pousse-jetons est en approche d'un *état stable*. Si des événements internes ou externes sont en attente de traitement, la procédure réalise les propagations correspondantes. Sinon, le pousse-jetons se bloque dans un *état stable* en attente d'événements déclencheurs qui viendraient le débloquer.

Les événements internes ont la particularité d'être surveillés entre chaque propagation à partir d'une place nouvellement marquée, car nous supposons que la dynamique interne du pousse-jetons est plus rapide que sa dynamique de communication. Nous souhaitons que la procédure d'exécution des RdPOC prenne en compte au plus tôt les événements internes, et en particulier les événements temporels. Les événements externes reçus par le pousse-jetons ne sont, eux, pris en compte qu'au moment de l'approche d'un *état stable*. L'émission d'événements externes est réalisée lorsque un jeton est contenu dans une place de sortie (elle-même alors contenue dans la liste `Pnm`). Un événement externe est créé à partir d'un couple constitué par la référence à la place de sortie et par le jeton émis. Il est envoyé par le pousse-jeton vers l'interface de communication, qui se chargera de router le message correspondant.

8.7 Conclusion et Remarques sur l'implantation

Le mécanisme d'exécution des RdPO du pousse-jetons, se veut efficace, en adoptant une approche par propagation et par recherche de l'état stable, qui permet d'éviter les exécutions cycliques, très coûteuses en terme de temps CPU. Il intègre également un mécanisme de gestion des priorités de tir des transitions (pour le déterminisme), un mécanisme de gestion du temps (pour gérer des fenêtres temporelles), un mécanisme de parallélisation des actions considérées comme longues et un mécanisme de verrouillage de chemins sur le graphe de contrôle d'un RdPO. La spécification complète du pousse-jetons est donnée dans [a1]. Le pousse-jetons étant la brique fondamentale du modèle d'exécution du langage CoSARC, il a été implanté dans le langage C++ afin de tester la validité de l'approche. Cette première implantation est actuellement portée sur le système temps-réel RT-AI (Real-Time Application Interface) qui fournit l'ensemble des primitives temps-réel nécessaires à la gestion des *timers* et *threads*. Plus globalement l'implantation des mécanismes des *conteneurs* (communication, référentiels, etc.) et du *gestionnaire de conteneurs* (ordonnancement) est également en cours sur ce même système temps-réel.

Le mécanisme d'exécution des RdPOC, supporté par l'entité appelée pousse-jetons, exécute les comportements asynchrones des composants rôles et des composants de contrôle. Or ces comportements sont eux-mêmes définis à partir de composants de représentation qui transitent dans les RdPOC. Ces composants de représentation peuvent être créés, assemblés et détruits dynamiquement,

au niveau du code d'un composant de contrôle ou d'un composant rôle. Ceci impose de pouvoir traduire ces opérations de modification sous la forme de code. Les composants de représentation sont, au niveau de leur implantation, considérés comme des objets spécifiques. Leurs descripteurs correspondant à des classes d'objets spécifiques (héritant d'une classe *Representation component* qui hérite elle-même d'une classe *object*). Chaque interface de représentation est traduite dans une interface objet classique. Un port de représentation fourni (typé par une interface) est traduit sous la forme d'une relation d'implantation de l'interface objet correspondante. Un port de représentation requis (typé par une interface) est traduit sous la forme d'un attribut qui est une référence sur l'interface objet correspondante. Une connexion de représentation correspond à la valuation d'un attribut représentant un port requis, par une référence sur un objet représentant un composant de représentation. Une connexion de représentation est instance d'une classe spécifique (héritant elle-aussi de *object*) qui possède une référence sur chacun des composants de représentation connectés, ainsi que sur l'attribut représentant le port de représentation requis concerné par la connexion. Les ports synchrones ne sont pas représentés. Les composants de représentation et leur assemblage est donc simplement traduit en terme de mécanismes objets. Grâce à cela, un RdPOC peut gérer le cycle de vie des composants de représentation (création, assemblage, désassamblage, destruction, création de jetons et stockage des composants de représentation dans le jetons, utilisation des services fournis). Inévitablement, nous devrons donc nous confronter au problème de l'allocation mémoire dynamique, classique dans le cadre des applications "temps-réel".

Le coté innovant de l'environnement d'exécution est qu'il propose des mécanismes qui facilitent les opérations de test. En premier lieu il rend plus facile le déploiement des composants en automatisant certaines procédures à partir des informations contenues dans les configurations. En second lieu, il offre un ensemble de services "techniques" qui permettent de contrôler l'exécution des composants, à différents niveaux (ordonnancement, assemblage, état interne, etc). Ce genre d'outil semble essentiel pour comprendre et "corriger" la dynamique des composants de contrôle. Notons que la mise en œuvre de cette approche devrait être favorisée par le fait que les éléments de description utilisés au niveau de l'environnement d'édition (e.g. les RdPO) existent et sont manipulables aussi au niveau de l'environnement d'exécution.

Parallèlement à l'implantation complète d'un prototype de gestionnaire de système CoSARC, qui demande beaucoup de travail avant d'être opérationnel, une architecture logicielle de contrôle d'un robot manipulateur a été, partiellement, modélisée à l'aide du langage CoSARC. Cet exemple d'architecture de contrôle a permis d'appliquer la méthodologie à un cas concret. Il est présenté dans le chapitre suivant.

Résumé :

L'environnement d'exécution des composants :

- gère le déploiement, l'ordonnancement, l'exécution et les communications des composants à partir de **conteneurs**.

- supporte un mécanisme d'exécution des réseaux de Petri à Objets (spécifié et implanté) :
  - basé sur une stratégie de construction des RdPO (décomposition des transitions).
  - basé sur la recherche d'un état stable.

- fournit aux développeurs un ensemble de services utiles pendant la phase de test, pour observer et contrôler l'exécution des composants de contrôle, des composants rôles et des conteneurs.

Chapitre 9

Exemple d'architecture d'un manipulateur mobile

Ce chapitre présente une modélisation de l'architecture d'un contrôleur de robot manipulateur mobile. Cette modélisation est faite en suivant la méthodologie CoSARC. Nous présenterons la phase d'analyse et la phase de conception et nous décrirons un déploiement en exemple. Des exemples de comportements asynchrones de composants de contrôle et de rôles, décrits par RdPO, sont également donnés à travers différents cas. Nous tenterons, à travers cet exemple de justifier les choix de conception fait pour un robot et illustrerons en cela l'apport de la méthodologie. Le robot manipulateur mobile considéré est schématisé dans la figure 9.1. Il est constitué d'un véhicule qui supporte un bras mécanique, une caméra motorisée et une antenne Wi-Fi omnidirectionnelle (pour les communications).

Le véhicule est non holonome. Il possède deux roues motrices (arrières) et deux roues directrices (avants). Il permet d'embarquer le reste de la partie opérative du robot ainsi que le contrôleur. L'instrumentation du véhicule est constituée d'un ensemble de capteurs et d'actionneurs. Une des roues arrières est équipée d'un *codeur incrémental* qui permet de connaître la vitesse du véhicule. Une des roues avant est équipée d'un *capteur de d'orientation* qui permet de connaitre l'orientation des roues et par là même en déduire la direction suivie par le véhicule. Quatre *émetteurs/récepteurs d'ultrasons* sont situés à l'avant du véhicule, ils permettent d'évaluer la distance entre le véhicule et un obstacle dans quatre directions données. Un *moteur* (commandé en vitesse) permet de propulser le véhicule. Un *servomoteur* (commandé en position) permet d'orienter les roues avant.

Le bras mécanique possède six articulations rotoïdes et six degrés de liberté au total. Sa base est placée à l'arrière du véhicule, son orientation initiale est définie dans le repère inverse du sens de marche du véhicule. Il embarque six *capteurs d'orientation* qui permettent d'évaluer la position angulaire de chaque articulation. Il embarque également six *moteurs* qui permettent d'orienter chaque articulation. En bout de bras, il est possible d'installer un effecteur dédié à une application donnée. Par exemple sur la figure 9.1, l'effecteur considéré est un préhenseur. Différents autres capteurs et actionneurs sont embarqués sur le véhicule. Une antenne Wi-Fi omnidirectionnelle permet au robot de communiquer via liaison hertzienne, elle permet également de connaître le niveau de réception. Une caméra motorisée est installée à l'avant du véhicule. Elle embarque un *capteur d'orientation* qui donne l'orientation de la caméra vis-à-vis du véhicule et un *servomoteur* qui permet de l'orienter. L'intervalle d'orientation est de [-90 °, +90 °], avec pour repère initial l'orientation dans le sens de la marche du véhicule. Enfin, un *capteur de niveau batterie* permet d'évaluer l'énergie restante du robot.

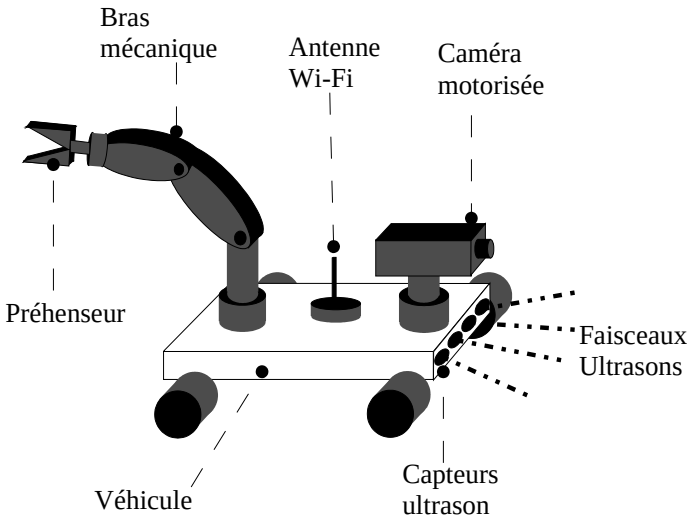


FIG. 9.1 – Schéma de la structure physique du robot manipulateur mobile

9.1 Phase analyse

La *première étape* de la phase d'analyse consiste à déterminer les capacités du robot, en terme de services qu'il doit pouvoir réaliser. Nous avons déterminé les capacités suivantes :

- se positionner par rapport à une carte de son environnement.
- identifier des caractéristiques de l'environnement : des objets de types connus, des murs, des couloirs.
- déterminer la position d'objets qui ont été désignés au robot par l'opérateur (via un mécanisme de pointage). Le robot est donc chargé de l'extraction du contour et de la détermination de la position.
- attraper des objets (d'un type connu) avec son bras et supporter la charge de l'objet.
- se déplacer dans l'environnement (de type structuré) ou laisser l'opérateur piloter le véhicule.

La contrainte à respecter est que le robot doit éviter les obstacles et qu'à aucun moment l'opérateur ne puisse obliger le robot à réaliser des ordres contradictoires ou qui entraîneraient sa destruction.

La *deuxième étape* consiste à représenter les éléments matériels constituant sa partie opérative. Les entités représentant les éléments matériels du robot manipulateur mobile sont un `véhicule`, un `bras mecanique`, une `caméra motorisée`, une `antenne Wifi` et une `batterie`. Chacune de ces entités peut être décomposée en un sous-ensemble d'entités représentant les éléments matériels qui la constitue. Par exemple, le `bras mécanique` peut-être décomposé en une `base`, des `segments`, des `articulations` et un `appendice terminal`. Le `véhicule` peut être décrit à partir d'entités telles qu'un `châssis`, des `essieux`, des `roues`, etc. A cette étape, on regroupe les propriétés physiques relatives à chaque élément matériel au sein des entités les représentant. Par exemple, les propriétés physique propres au bras mécanique sont regroupées au sein de l'entité `bras mécanique` ou des entités représentant les éléments matériels le constituant : la longueur des segments du `bras mécanique` sur les entités `segments`, le degré de battement sur les `articulations`, etc. La figure 9.2 présente un modèle UML simplifié qui répertorie la décomposition du `véhicule` en éléments matériels. Pendant cette phase sont aussi répertoriés les capteurs et actionneurs (physiques). Les `émetteurs/récepteurs ultrasons`, le `capteur d'orientation direction` (du véhicule), le `capteur de vitesse angulaire` (du véhicule) sont ainsi des exemples de capteurs représentés, ils

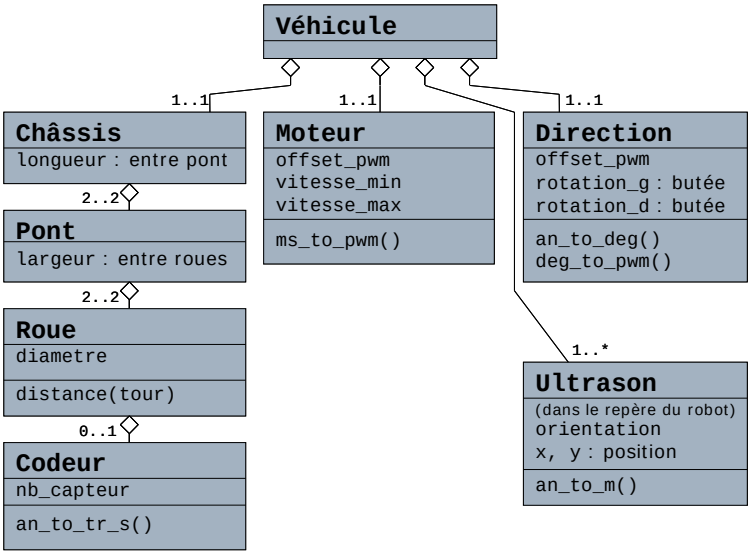


FIG. 9.2 – Modèle UML des éléments matériels constitutifs du véhicule

sont des constituants de l'entité `véhicule`. Chaque type d'élément matériel peut être représenté, pendant la phase d'analyse, par une classe (cf. fig. 9.2). Par exemple, une classe représente l'élément matériel `bras mécanique`, elle possède des méthodes de calcul de ses modèles géométriques et cinématiques (direct et inverse). L'entité représentant le `véhicule` possède une méthode de calcul de son déplacement qui se base sur les propriétés physique du véhicule.

La *troisième étape* consiste à déterminer les boucles de réactivité "directes" utilisées pour commander les éléments matériels identifiés. Pour cela, nous avons établi un ensemble de lois de commande disponibles pour contrôler le robot. Dans notre exemple, une seule loi de commande est utilisée pour contrôler les déplacements du véhicule. Le contrôle du bras manipulateur est basé sur cinq lois de commande différentes et complémentaires [34]. Ces lois de commande sont brièvement présentées dans dans les sections suivantes.

La *quatrième étape* consiste alors à synthétiser l'analyse en déterminant les **ressources** présentes dans le contrôleur. Dans l'exemple présenté nous avons identifié trois ressources : le `Mobile`, le `Manipulateur` et la `Vision`. La `Vision` est la ressource chargée de déterminer la position des objets désignés par l'opérateur dans l'environnement (on suppose dons que cette ressource s'appuie sur un algorithme ou une technlogie de caméra particulière pour reconstruire la profondeur). Elle permet également d'identifier certaines caractéristiques de l'environnement (e.g. murs). Le `Mobile` est la ressource chargée des déplacements du robot et des évitements d'obstacles. Le `Manipulateur` est la ressource chargée de la saisie et du transport de la charge. Les mouvements du véhicule et de la caméra peuvent être couplés lorsque, par exemple, le robot doit se positionner sur la carte de l'environnement. En effet, les mouvements du véhicule vont changer le positionnement de la caméra qui devra prendre en compte ce mouvement. Dans ce cas, il pourrait être nécessaire de faire émerger une quatrième ressource nommée `VisionMobile` qui couple le contrôle du véhicule et de la caméra. Néanmoins, dans un souci de simplicité, nous ne retenons pas cette solution.

Pour chacune des trois ressources, nous définissons leurs différents modes. Le `Manipulateur` fonctionne dans un unique mode autonome dans lequel il permet au bras mécanique de réaliser : un

déplacement vers une position dans l'espace, une entrée en contact avec une surface, la saisie d'un objet, le déplacement d'un objet situé à une position donnée vers une position cible (ces deux positions se trouvant dans l'espace de travail du bras), le support d'une charge pour un transport amorti. La `Vision` fonctionne dans deux modes : le mode autonome dans lequel elle est capable d'indentifier et de positionner des objets et d'autres caractéristiques dans l'environnement perçu ; le mode téléopéré dans lequel l'opérateur peut commander les déplacements de la caméra, reçevoir un retour visuel et demander l'identification (via un mécanisme de pointage) d'objets dans l'environnement (dont on suppose ici qu'elle sait reconnaître la forme) et leur mémorisation dans la carte de l'environnement. Enfin, le `Mobile` peut fonctionner dans deux modes : le mode autonome dans lequel le véhicule est capable de réaliser des déplacements dans l'environnement, en garantissant l'évitement d'obstacles ; le mode téléopéré qui permet à l'opérateur de piloter le véhicule à distance.

En fonction des boucles de réactivité "directes" disponibles et des services réalisés par chaque mode, il est alors nécessaire de déterminer les actions qu'une ressource est capable de réaliser. Par exemple, la ressource `Mobile` est capable de réaliser deux actions : "opérateur pilote véhicule" via laquelle le véhicule est directement asservi par des consignes provenant d'un opérateur humain ; "suivre un chemin" qui permet au véhicule d'atteindre en autonome un point dans l'environnement (si possible), tout en évitant des obstacles. La définition des *actions* d'une ressource peut reposer sur différentes lois de commandes. Une entité *action* peut donc interagir avec une ou plusieurs entités *commande*. Par exemple, une unique loi de commande en position est utilisée au sein de la ressource `Mobile`, elle est appliquée par une entité *commande* unique. Les deux entités *actions* présentes dans le `Mobile` utilisent cette *commande*. Les *actions* du `Manipulateurs` sont définies à partir de plusieurs lois de commande. Chacune de ces lois de commande est adaptée à l'établissement d'un comportement particulier du bras mécanique (e.g. positionnement, recherche de contact, suivi de surface) dans un espace donné (effort, position cartésienne ou articulaire, etc.) et possède un domaine de validité qui lui est propre et en dehors duquel elle n'est pas applicable. Chacune des *actions* que peut effectuer le `Manipulateur` est réalisée à travers l'application d'une ou plusieurs de ces lois, et supporte de fait un mécanisme de commutation entre lois de commande [36]. En conséquence, les entités *action* interagissent chacune avec plusieurs entités *commande* qui applique chacune une loi de commande spécifique.

A partir de cette analyse du contrôleur de robot manipulateur mobile, nous raffinons sa description afin d'obtenir une modélisation de l'architecture en terme de composants de contrôle, connecteurs, configurations et composants de représentation.

9.2 Architecture logicielle de contrôle du manipulateur mobile

L'architecture logicielle de contrôle du manipulateur mobile est présentée dans la figure 9.3. La configuration nommée `ContrôleurManipulateurMobile` englobe cette architecture et elle est en interaction avec un composant `Plate-forme Opérateur`. Les trois ressources sont représentées sous la forme de trois configurations connectées au composant de contrôle `SuperviseurGlobalManipulateur Mobile`. Ce dernier est également connecté à un composant de contrôle `GénérateurEvénementsBatterie` (qui lui permet d'être notifié d'un manque d'énergie) et à un composant de contrôle `GénérateurEv -énementsWifi` (qui lui permet de savoir si le lien de communication avec l'opérateur n'est pas rompu). Le `ContrôleurManipulateurMobile` possède un ensemble d'attributs qui représentent tous les éléments matériels constitutifs du robot manipulateur mobile, tous les capteurs et actionneurs embarqués, ainsi que toutes les lois de commandes utilisées. A partir de ces attributs, le `ContrôleurManipulateurMobile` fixe les valeurs des attributs configurables des différents composants de contrôle et configurations contenus, au moment de leur instanciation (i.e. au moment de l'appel de leur méthode d'initialisation). Par exemple (cf. fig. 9.3), il paramètre la ressource `Manipulateur` en fonction (entre autres) d'un composant de représentation `BrasMécanique` et il paramètre le `ContrôleurE/SMicrocon-`

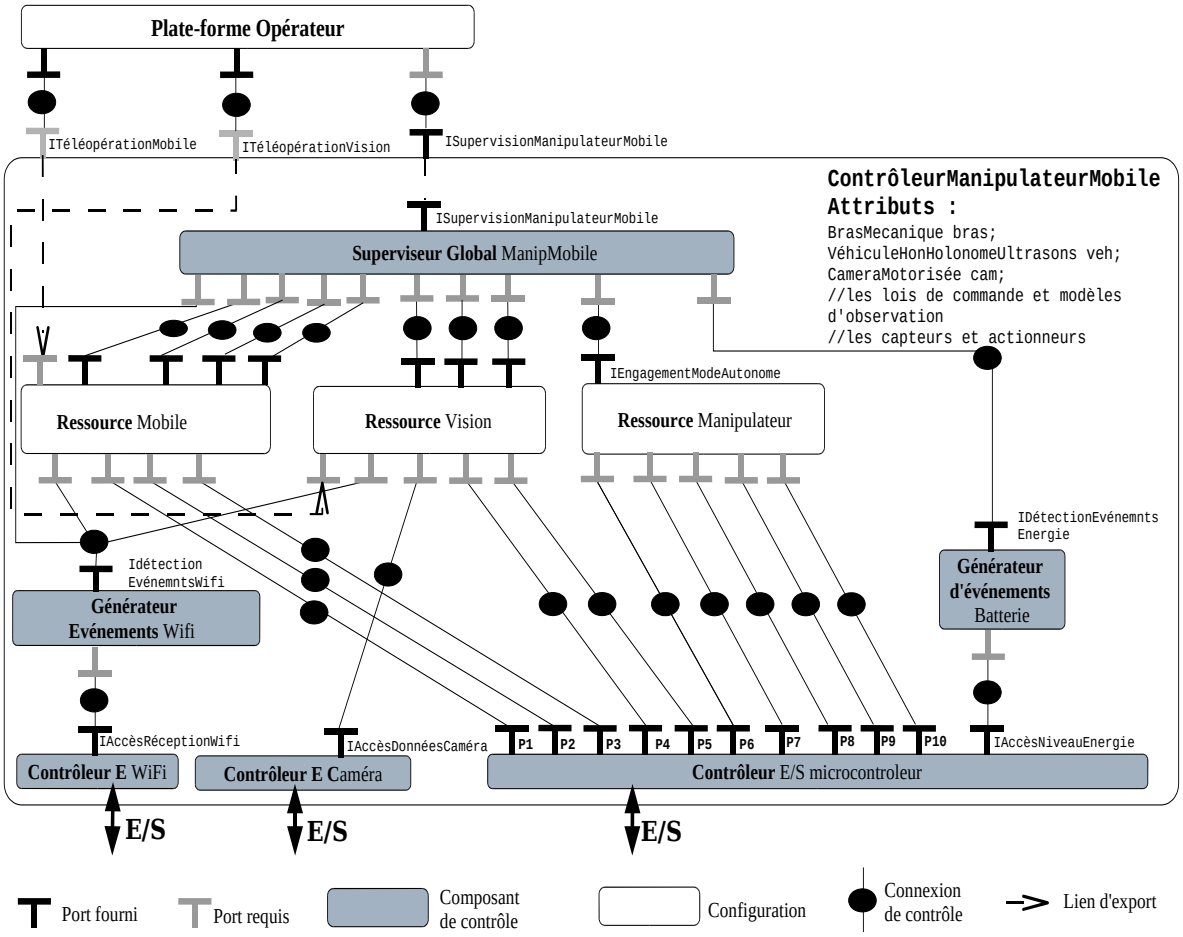


FIG. 9.3 – Architecture du contrôleur d'un robot manipulateur mobile

`trôleur` en fonction (entre autres) d'un composant de représentation `CapteurMouvementsAngulaire-BrasMécanique`.

Il existe dans cette architecture un ensemble de contrôleur d'entrées/sorties qui permettent d'accéder aux différents capteurs et actionneurs disponibles pour contrôler le robot. Le `ContrôleurEWifi` permet d'avoir accès aux informations issues du `CapteurNiveauRéceptionWifi` (via son port fournissant le service `IAccèsRéceptionWiFi`). Le `ContrôleurECaméra` permet d'avoir accès aux données produites par la caméra (via son port fournissant le service `IAccèsDonnéesCaméra`). Le `ContrôleurE/S Microcontrôleur` est un composant de contrôle spécifique créé pour dialoguer avec un microcontrôleur via un port série. Ce microcontrôleur est relié à un ensemble de capteurs et d'actionneurs. Il permet d'avoir accès à toutes les données issues du `CapteurNiveauEnergie`, du `CapteurVitesseVéhicule`, du `CapteurOrientationVehicule`, du `CapteurOrientationCaméra`, des `CapteursUltrasons`, des `CapteursOrientationAngulaireBrasMécanique` (valeur angulaire des articulations), du `CapteurEffortPince` (effort en bout de pince). Le `ContrôleurE/SMicrocontrôleur` permet également d'avoir accès à tous les actionneurs : le `MoteurVéhicule` ; le `ServoMoteurOrientationVéhicule` ; les six `MoteursArticulationsBrasMécanique` ; le `MoteurPince` ; le `MoteurOrientationCaméra`.

Les contrôleurs d'entrées/sorties proposent des services d'accès aux données capteurs et actionneurs sous une forme standardisée (indépendante, si possible, du matériel), aux ressources (à travers leurs ports fournis), au superviseur global et aux générateurs d'événements de l'architecture. Par

exemple le `ContrôleurE/SMicrocontrôleur` possède un ensemble de ports fournis (cf. fig. 9.3) qui donne accès aux services décrits par les interfaces de contrôle suivantes : `IAccèsNiveauEnergie` ; `IAccès OrientationCaméra` (P4) ; `IAccèsMesureUltrasons` (P1) ; `IAccèsOrientationEtVitesseVéhicule` (P3) ; `IAccèsEffortPince` (P8) ; `IAccèsPositionAngulaireBras` (P7) ; `IAccèsMoteursDirectionEtVitesseVéhicule` (P2) ; `IAccèsMoteurOrientationCaméra` (P5) ; `IAccèsMoteursArticulationsBras` (P6) ; `IAccèsMoteurPince` (P9) ; `IAccèsFreinsBras` (P10).

Les attributs des trois contrôleurs d'entrées/sorties sont des composants de représentation décrivant les capteurs et/ou les actionneurs avec lesquel ils dialoguent. Ils ont également pour attributs les composants de représentation décrivant les éléments matériels (par exemple le `Véhicule`) sur lesquels les capteurs et actionneurs sont embarqués. Les composants de représentation *capteurs* et *actionneurs* servent à stocker toutes les informations relatives aux périphériques correspondants et leurs ports fournis permettent, par exemple, de fixer la dernière donnée échantillonée (pour un capteur) ou la dernière donnée de commande appliquée (pour un actionneur) ou de connaître la date de dernière modification. Leur ports requis sont connectés aux ports fournis des composants de représentation décrivant les éléments matériels. Par exemple, un port requis du `CapteurOrientationCaméra` est connecté à un port fourni de l'élément matériel `Caméra` (afin de mettre à jour sa valeur d'orientation). Ceci permet par exemple à un composant représentant un capteur, de mettre "automatiquement" à jour la valeur de position de l'élément matériel correspondant, lors de l'utilisation de son service d'acquisition.

Chaque ressource est reliée aux contrôleurs d'entrées/sorties qui fournissent les services de bas niveau qu'elle requiert. Par exemple, la `Vision` est connectée au `ContrôleurE/SMicroContrôleur` pour pouvoir envoyer des données de commande aux moteurs de la caméra et récupérer les données d'orientation de la caméra. Elle est également connectée au `ContrôleurECaméra` qui lui permet d'avoir accès aux données extraites de l'image (données provenant par exemple d'un composant électronique FPGA [126]). Enfin, elle est connectée au composant de contrôle `GénérateurEvénementsWifi` qui lui fournit les services d'accès au "niveau de réception" Wifi, information utile, dans cet exemple, lorsqu'une téléopération directe est établie.

Une ressource est décrite via une configuration constituée de multiples composants de contrôle. Les ports requis des composants *ressource* sont pour la plupart, des ports d'accès au services offerts par les contrôleurs d'entrée/sortie. Les ports fournis sont essentiellement des offres de services liés à la gestion des modes. Puis la ressource `Manipulateur` n'a qu'un mode (autonome), il ne dispose que d'un seul port fourni qui permet offre un service d'activation de ce mode (i.e. réalisation d'un ordre de ressource en autonome). La ressource `Vision` et la ressource `Mobile` possèdent deux ports pour contrôler l'activation de chaque mode et un port permettant de désengager le mode courant. Il n'est pas envisageable pour des raisons de longueur de présentation de cet exemple, de décrire complètement toutes ces ressources. Nous présenterons par la suite la ressource `Mobile` et la ressource `Manipulateur` et nous montrerons le détail de certains des composants qui les constituent.

9.3 Ressource Mobile

Le véhicule est la partie matérielle sur laquelle repose tout le reste de l'équipement embarqué. Nous voulons, dans un premier temps, tester la méthodologie CoSARC sur un matériel relativement simple à contrôler tel que ce véhicule. Nous pourrons dans le futur faire évoluer l'architecture du contrôleur au fur et à mesure que sa partie opérative est améliorée et enrichie, afin d'obtenir l'architecture globale définie dans la section précédente. Nous partons donc d'un véhicule non-holonome simple (cf. fig. 9.4) pour lequel nous développons un contrôleur.

Ce véhicule possède les caractéristiques physiques suivantes :

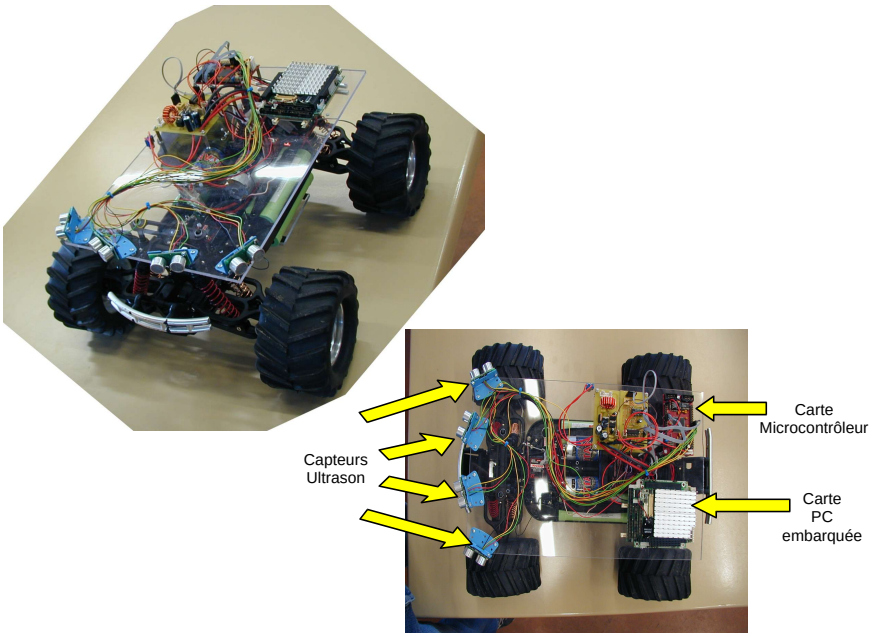


FIG. 9.4 – Le véhicule sur lequel est basé le développement

- Le codeur incrémental permettant de calculer la vitesse du véhicule ne génère qu'un top par tour de roue, la précision du capteur est donc d'un tour de roue soit 22 cm.
- L'orientation des roues avants est comprise entre [-30 °,+30 °] et l'intervalle d'angle sur lequel le servomoteur permet de contrôler l'orientation des roues avants est le même.
- L'intervalle de distance sur lequel les capteurs ultrasons sont opérationnels est de [0m, 4m].
- Les vitesses maximales du moteur du véhicule sont de 12 tours/s en marche arrière et de 15 tours/s en marche avant.
- Un microcontrôleur est chargé de piloter l'ensemble des capteurs et actionneurs embarqués à l'exception de l'antenne Wi-Fi. Une carte PC embarquée permet d'exécuter le contrôleur. Le microcontrôleur et la carte PC embarquée dialoguent via une liaison série.

9.3.1 Architecture interne

L'architecture logicielle modélisée pour le cas d'un robot mobile est présentée dans la figure 9.5. Elle ne contient qu'une seule ressource : le `Mobile`. Le `SuperviseurGlobalRobotMobile` est en charge de gérer l'utilisation de cette ressource, en fonction de l'énergie disponible. Pour cela le `SuperviseurGlobalRobotMobile` possède un port requis `P2` via lequel il est lié à un composant `G.E.Batterie` fournissant le service de détection d'événements liés au niveau d'énergie. Les ports requis `P3` à `P6` sont connectés aux ports fournis par la ressource `Mobile` (`P7` à `P10`) : ils permettent de contrôler l'engagement et le désengagement des modes du `Mobile`. Le port fourni `P1` est connecté avec la plateforme opérateur, d'où le `Mobile`. Le `SuperviseurGlobalRobotMobile` reçoit des ordres de mission à exécuter (destinés au mode téléopéré ou au mode autonome). Les couches basses contiennent le `contrôleurE/SMicrocontrôleur` et le `contrôleurEWiFi`, tous deux chargés de fournir au reste de l'architecture des services d'accès aux capteurs/actionneurs. Le `contrôleurE/SMicrocontrôleur` permet d'accéder à tous les capteurs et actionneurs, à l'exception de l'antenne Wi-Fi, dont le niveau de réception est contrôlé par le `contrôleurEWi-Fi`. Chaque port (fourni) du `contrôleurE/SMicrocontrôleur` (`P61` à `P65`) correspond à un service d'accès pour un ensemble de capteurs ou d'actionneurs.

Par exemple, le port P61 fournit un service d'accès au niveau d'énergie mesuré par le capteur de la batterie (interface IAccèsNiveauEnergie) ; le port P63 fournit le service d'accès au moteur et au servomoteur du véhicule (interface IAccèsMoteursDirectionEtVitesseVéhicule) ; le port P64 fournit le service d'accès aux mesures des quatre capteurs à ultrasons (interface IAccèsMesureUltrasons). Les deux générateurs d'événements G.E. Batterie et G.E. Wi-fi peuvent être configurés pour détecter le dépassement de seuils sur les données capteurs de la batterie (seuil d'énergie faible) et de l'antenne Wi-Fi (perte de signal ou signal de réception faible).

La ressource Mobile possède dans son architecture interne, l'ensemble des composants de contrôle suivants :

- Le SuperviseurRessourceMobile est chargé de gérer la cohérence d'utilisation des modes (un seul mode actif à la fois). Il est également chargé de la commutation des modes en assurant que le véhicule reste stable pendant cette commutation. Il interagit avec les deux modes afin de réaliser les ordres provenant du SuperviseurGlobalRobotMobile. Ses ports P7 à P10 fournissentt les services liés à l'engagement et au désengagement des modes.
- Le ModeVéhiculeTéléopéré permet de gérer une téléopération directe du Véhicule par un opérateur humain. Son port P21 fournit le service IModeTéléopérationVéhicule.
- Le ModeVéhiculeAutonome permet de gérer les déplacements du Véhicule en fonction d'ordres de haut niveau d'abstraction (e.g. "aller-à", "déplacer-de-vers"). Son port P19 fournit le service IModeAutonomeVéhicule.
- L'ActionVéhiculeSuivreCheminAvecEvitementObstacles permet de réaliser tous les déplacement autonomes du véhicule dans l'environnement (consignes de bas-niveau). Tous les ordres de ressource reçus par le ModeVéhiculeAutonome se traduisent par l'utilisation (éventuellement répétitive) de ce composant de contrôle. Elle encapsule un mécanisme d'évitement d'obstacles basé sur une *zone virtuelle déformable*. Son port P30 fournit le service IActionVéhiculeSuivreChemin.
- L'ActionOpérateurPiloteVéhicule permet à un opérateur de téléopérer directement le véhicule afin de contrôler ses déplacements. Elle gère les communications (en particulier les retards de communication entre la plate-forme opérateur et le Mobile) et assure aussi que le véhicule ne percutera pas d'obstacles. Son port P32 fournit le service IActionOperateurPiloteVéhicule.
- Le GénérateurEvénementObstacle permet de notifier la présence d'un obstacle dans la trajectoire du véhicule. Il se base pour cela sur les données issues des capteurs à ultrasons. Son port P48 fournit le service IDétectionObstaclesVéhicule.
- La PerceptionPositionVéhicule permet à tout instant de calculer la position du véhicule en fonction des données du capteur d'orientation des roues avants et du capteur de vitesse du véhicule (et à partir d'une orientation et d'une position initiale du véhicule dans l'espace). Son port P47 fournit le service IPerceptionPositionVéhicule.
- La PerceptionEnveloppeVéhicule permet de reconstituer l'enveloppe virtuelle courante du véhicule, elle-même calculée en fonction des données issues des capteurs ultrasons. Elle est utilisée par l'ActionVéhiculeSuivreCheminAvecEvitementObstacles pour calculer la déformation de l'enveloppe virtuelle (causée par la présence d'un obstacle) et ainsi, en déduire la réaction à appliquer. Son port P44 fournit le service IPerceptionEnveloppeVéhicule.
- La CommandeVéhiculePosition permet d'asservir le véhicule à partir d'une suite de positions dans l'espace. Ce composant de contrôle est utilisé par les deux composants action présents dans la ressource Mobile. Son port P45 fournit le service ICommanderVéhiculePosition.

Cette première ébauche de l'architecture de la ressource Mobile permet de mettre en relief les dépendances métier entre composants de contrôle, exprimées au travers de services fournis et requis via leurs ports de contrôle. Nous avons choisi de définir un seul composant *commande*, utilisé par les deux composants *action* de la ressource. Néanmoins, une approche différente, basée sur deux composants commande différents (celui présenté utilisé par l'action de téléopération et un autre com-

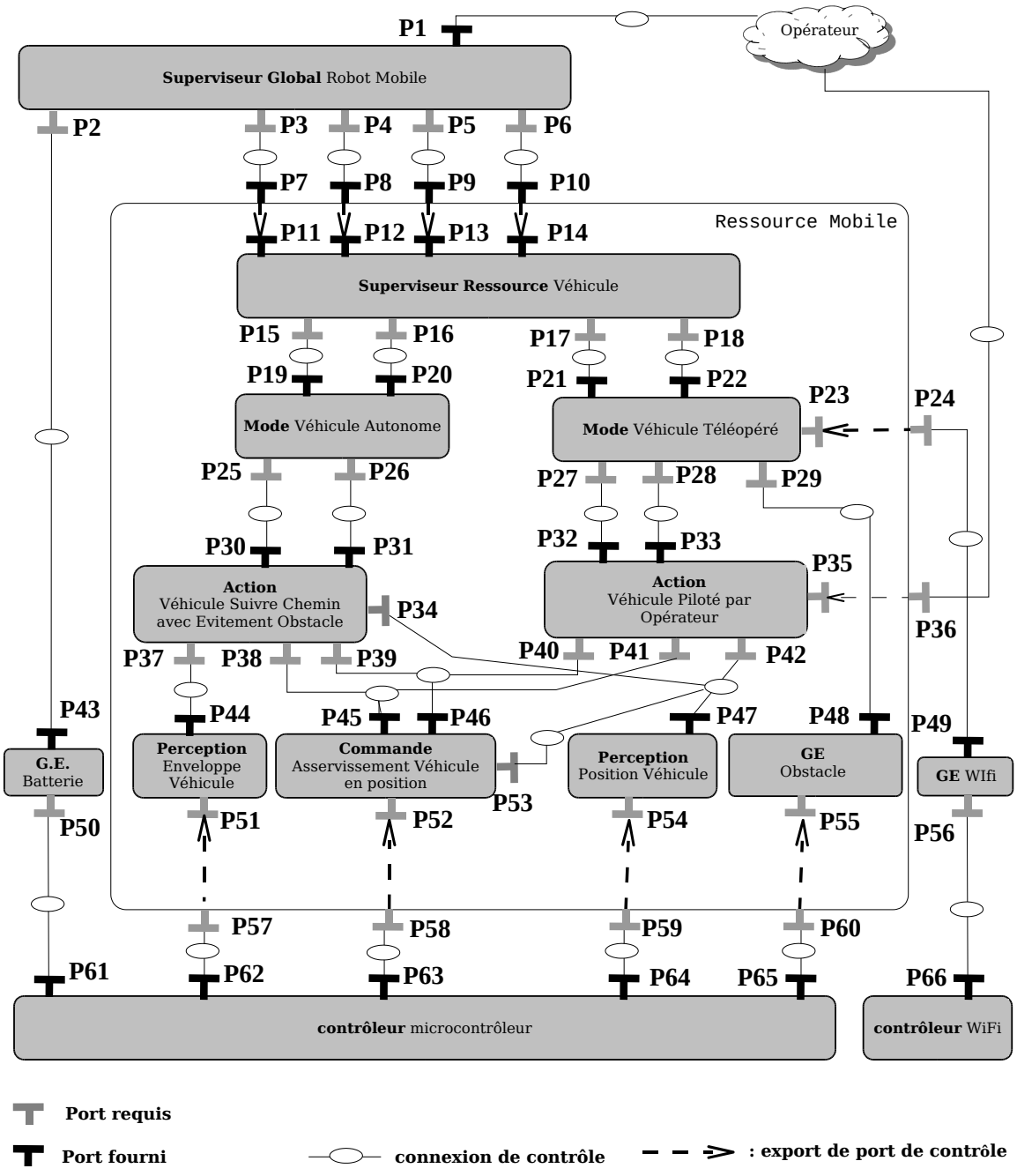


FIG. 9.5 – Architecture de la ressource `Mobile` : composants de contrôle et configuration

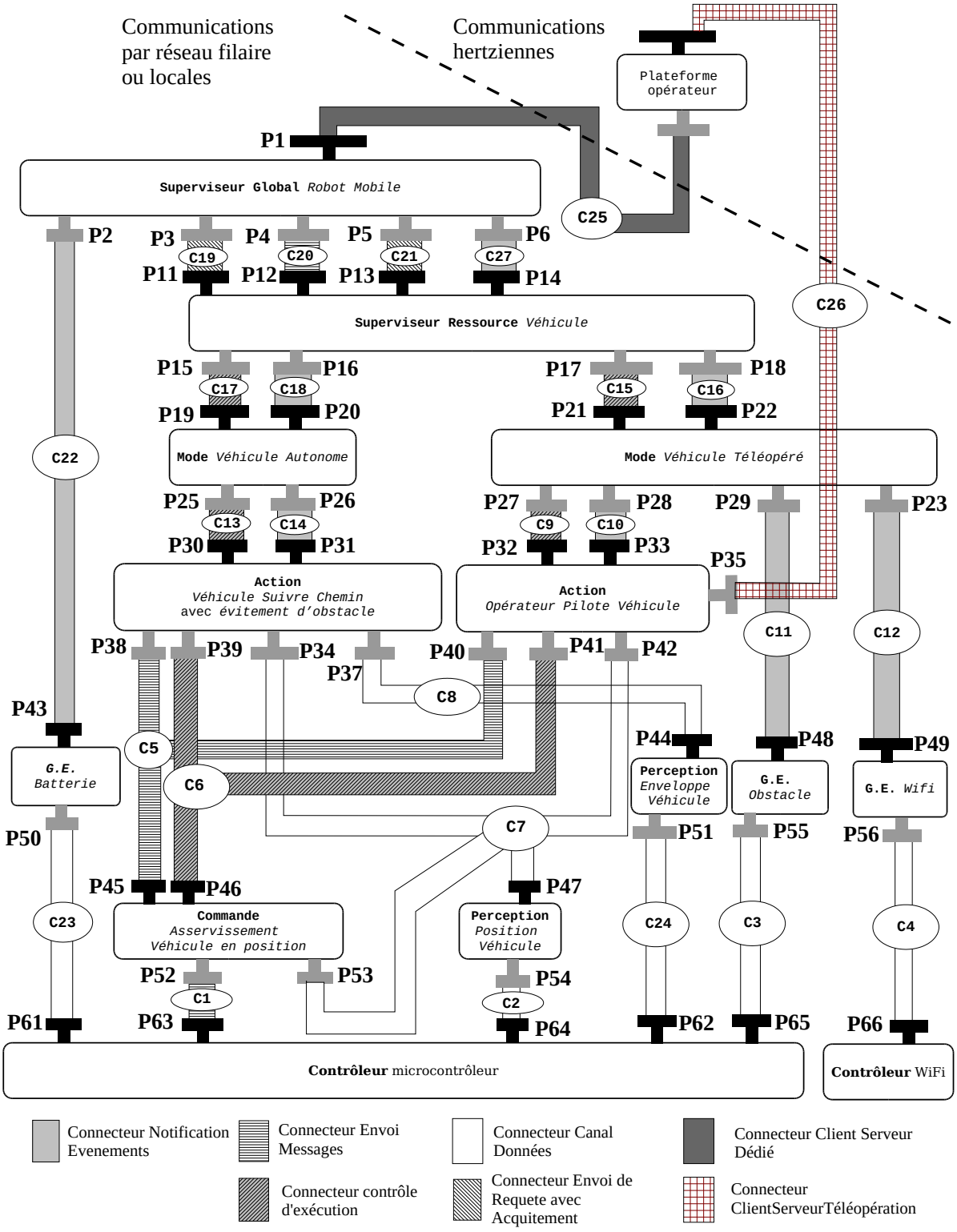


FIG. 9.6 – Architecture de la ressource `Mobile` : connecteurs

posant commmande intégrant directement la zone virtuelle déformable, utilisé par l'action autonome de déplacement), pourrait se révéler pertinente dès lors que l'on considère l'efficacité à l'exécution.

9.3.2 Connecteurs

Les différents descripteurs de connecteurs utilisés sont présentés dans la légende de la figure 9.6. Il représentent chacun un protocole d'interaction entre composants de contrôle qui peut être réutilisé à travers différentes architectures de contrôle.

- Le descripteur `NotificationEvénement` définit un protocole de notification d'événements. Un événement est le nom que nous donnons à un composant de représentation qui représente un phénomène sporadique, détecté à partir d'un critère. Le composant `RôleNotifieur` est chargé d'enregistrer les abonnés à l'écoute d'un événement et de notifier l'occurrence de cet événement aux composants `RôleEcouteur`. Un composant `RôleEcouteur` est chargé de s'abonner auprès du composant `RôleNotifieur`. Lorsqu'un événement est émis, le composant `RôleNotifieur` désenregistre automatiquement ses abonnés et les composants `RôleEcouteurs` ne sont plus sensibles à l'événement une fois qu'il a été reçu. Des connecteurs `NotificationEvénement` sont typiquement utilisés pour les interactions entre les générateurs d'événements et les autres composants de contrôle. Les connecteurs `c11`, `c12` et `c22` (cf. fig. 9.6) sont des exemples de connecteurs instances de ce descripteur.
- Le descripteur `EnvoiMessage` définit un protocole d'envoi asynchrone de message sans attente de réponse. Un à plusieurs composants `RôleEmetteur` envoient des messages et un composant `RôleReceveur` reçoit le message. Ce descripteur est typiquement utilisé pour les interactions entre les composants commande et les contrôleurs d'entrées/sorties (envoie des données de commande). Par exemple, le connecteur `c1` (cf. fig. 9.6) est instance de ce descripteur.
- Le descripteur `ContrôleExécution` permet de mettre en place une interaction dans laquelle un ou plusieurs composants de contrôle jouant le rôle `Maître` contrôlent l'exécution d'un unique composant de contrôle jouant le rôle `Exécuteur` : dès qu'un composant `RôleMaître` a demandé le démarrage de l'exécution du composant `RôleExécuteur`, ce dernier ne peut plus recevoir de demande de démarrage d'un autre composant `RôleMaître`, jusqu'à ce que le `RôleMaître` (en cours) ait demandé l'arrêt de l'exécution. Le composant jouant le rôle `Maître` peut également faire la mise à jour de l'exécution dans le cas où il fait de nouvelles demandes de démarrage (considérées comme des mises à jour des paramètres d'exécution). Des connecteurs de ce type sont utilisés pour connecter différents composants constituant une ressource (cf. fig. 9.6) pour assurer qu'un seul composant contrôle l'exécution, à un moment donné, du ou des composants de contrôle d'une couche inférieure. Par exemple, les deux composants *action* interagissent avec le composant *commande* via le connecteur `c6` (cf. fig. 9.6) qui est une instance de ce descripteur.
- Le descripteur `CanalDonnées` définit un protocole d'enregistrement à un flux de données. Dans ce protocole des `RôlesSouscrivants` s'enregistrent auprès d'un `RôlePublieur` qui publie des données. Le désenregistrement ne peut être le fait que des composants de contrôle jouant le rôle `Souscrivant`. Des connecteurs `CanalDonnéess` sont typiquement utilisés pour les interactions entre les composants de contrôle *perception* et les autres composants de l'architecture logicielle. Par exemple (cf. fig. 9.6), les connecteurs `c8`, `c12` et `c24` sont des exemples d'instances de ce descripteur.
- Le descripteur de connecteur `EnvoiRequêtesAvecAcquitement` encapsule un protocole de type Requête/Réponse (cf. chapitre 7) dans lequel la réception de la requête par le `RépondeurAvecAcquitement`, est confirmée par un acquitement. La réponse contient le résultat de la terminaison du traitement de la requête. Des connecteurs `EnvoiRequêtesAvecAcquitement` sont utilisés pour les interactions entre le superviseur global et le superviseur de la ressource `Mobile` : le superviseur global reçoit un acquitement dès lors que sa requête a été reçue par le `SuperviseurMobile` ; il reçoit le compte-rendu d'exécution de la requête lorsque le `SuperviseurMobile` a réalisé

les traitements correspondants à la requête. Les connecteurs `c19` et `c21` (cf. fig. 9.6) sont des exemples d'instances de ce descripteur.
- Le descripteur de connecteur `ClientServeurDédié` encapsule une interaction client/serveur dédiée, le rôle `ServeurDédié` n'admettant plus de communication avec d'autres `Clients` tant que le premier `Client` n'est pas déconnecté (n'acceptant qu'une seule connexion, au sens TCP). Un connecteur `ClientServeurDédié` est utilisé pour connecter le superviseur global à la plateforme opérateur (e.g. `c25`, cf. fig. 9.6)
- Le descripteur de connecteur `ClientServeurTéléopération` encapsule un protocole d'échange de données entre deux rôles : `EmetteurConsignes` et `EmetteurDonnées`. Ces deux rôles échangent des données afin de réaliser le lien de téléopération suivant un protocole trinomial sur UDP. Un connecteur `ClientServeurTéléopération` est utilisé pour les échanges entre l'`ActionOpérateur` `PiloteVéhicule` et la plateforme opérateur, au moment d'une téléopération directe du véhicule (e.g. `c26`, cf. fig. 9.6).

Deux descripteurs de connecteurs sont présentés en exemple par la suite : `ContrôleExécution` et `NotificationEvénement`.

9.3.3 Descripteur `ContrôleExécution`

La figure 9.7 présente un descripteur de connecteur `ContrôleExécution`. `ContrôleExécution` est décrit à partir de deux descripteurs de composants rôles (`RôleMaître` et `RôleExécuteur`). `RôleMaître` est le composant rôle à travers lequel un composant de contrôle prend le contrôle de l'exécution d'un autre composant de contrôle jouant le rôle `Exécuteur`. Le descripteur de `RôleMaître` possède trois descripteurs de ports asynchrones fournis `PAs1`, `PAs2`, `PAs3`.
- `PAs1` référence l'interface `In démarrer(anyPD)`. Ce descripteur de port décrit un point de réception des messages de demande de démarrage d'exécution du composants jouant le role `Maître`, ou de mise à jour de ses paramètres d'exécution. Le paramètre `anyPD` peut être d'un type de composants de représentation quelconque (ou une collection de types). Il correspond aux paramètres d'exécution passés (par la suite) au composant rôle `RôleExécuteur`.
- `PAs2` référence l'interface `In arrêter()`. Il décrit le point de réception des demandes d'arrêt d'exécution envoyées à un composant de contrôle jouant le rôle `Exécuteur`.
- `PAs3` référence l'interface `Out arrêté(anyPA)`. Il décrit un point d'émission des confirmations d'arrêt, via lequel un composant de contrôle jouant le rôle `Maître` prend connaissance des comptes-rendus d'exécution lors de l'arrêt de celui jouant le rôle `Exécuteur`.

L'interface de rôle `Maître`, référencée par le descripteur de port fourni du descripteur `ContrôleExécution`, contient ces trois interfaces asynchrones (cf. fig. 9.7).

Le descripteur de composant rôle `RôleExécuteur` possède trois descripteurs de port asynchrones fournis `PAs4`, `PAs5` et `PAs6`.
- `PAs4` référence l'interface `Out démarrer(anyPD)`. Il décrit le point de réception de message de démarrage (ou de mise à jour) d'exécution, d'un composant de contrôle jouant le rôle `Exécuteur`.
- `PAs5` référence l'interface `Out arrêter()`, il décrit le point de réception de message d'arrêt d'exécution, pour un composant de contrôle jouant le rôle `Exécuteur`.
- `PAs6` référence l'interface `In arrêté(anyPA)`. Il décrit le point d'émission des messages de confirmation d'arrêt, d'un composant de contrôle jouant le rôle `Exécuteur`.

L'interface de rôle `Exécuteur`, référencée par le descripteur de port requis de `ContrôleExécution`, contient ces trois interfaces asynchrones (cf. fig. 9.7). Dans le protocole de `ContrôleExécution`, un composant de contrôle joue le rôle `Exécuteur` et un ou plusieurs composants de contrôle jouent le rôle `Maître`.

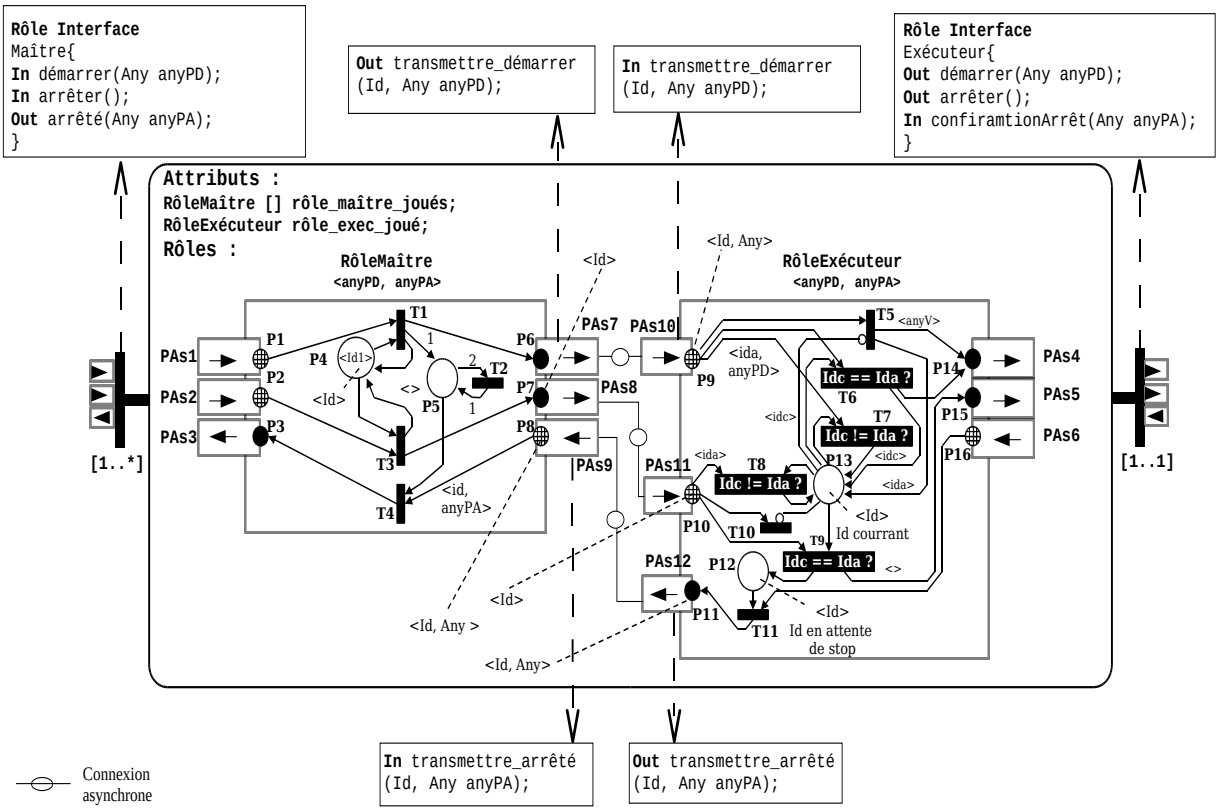


FIG. 9.7 – Représentation graphique du descripteur de connecteur `ContrôleExécution`

Les comportements asynchrones liés aux deux rôles sont présentés dans la figure 9.7. Lorsqu'un jeton arrive dans la place d'entrée `P1` associée au port `PAs1`, la transition `T1` est tirée : un jeton banalisé est placé dans la place `P5` afin de mémoriser l'activité du composants rôle `RôleMaître` ; un jeton contenant les paramètres d'exécution ainsi que l'identifiant du `RôleMaître` émetteur est placé dans la place de sortie `P6` (liée au port requis `PAs7`). Si le composant jouant le rôle `RôleMaître` émet à nouveau des demandes de démarrage, elles seront considérées comme des mises à jour des paramètres d'exécution (tir de `T1` correspondant à la demande de mise à jour, puis tir de `T2` afin de maintenir un unique jeton dans `P5`). Lorsque le jeton est émis par la place de sortie `P6`, il est routé via le port `PAs7` vers le port `PAs10` du composant rôle `RôleExécuteur`. Le jeton arrive alors dans la place d'entrée `P9`. Si le `RôleExécuteur` n'a pas démarré l'exécution du composant de contrôle auquel il est rattaché (celui qui joue le rôle `Exécuteur`, il le fait (tir de `T5`). S'il a déjà démarré l'exécution du composant de contrôle jouant le rôle `Exécuteur` et que l'émetteur du message est celui qui a déjà demandé l'exécution, les données transmises sont alors considérées comme des paramètres de mise à jour (tir de `T6`). Dans ces deux cas, un jeton est placé dans la place `P13` après le tir de l'une ou l'autre transition. Si un jeton est reçu dans `P9`, que le composant rôle `RôleExécuteur` a déjà enregistré un composant rôle `RôleMaître` et que l'identifiant de ce `RôleMaître` est différent de celui du `RôleMaître` ayant demandé l'exécution, alors le jeton est supprimé (tir de `T7`). La demande n'est donc pas prise en compte, car un `RôleMaître` a déjà le contrôle de l'exécution du `RôleExécuteur`.

Lorsqu'un composant jouant le rôle `Maître` fait une demande d'arrêt, un jeton est placé dans la place d'entrée `P2` du composant rôle `RôleMaître` correspondant, ce qui amène au tir de la transition `T3`, puis au dépôt d'un jeton dans sa place de sortie `P7`. Le jeton est alors routé entre les ports `PAs8` et `PAs11`, et il est déposé dans la place d'entrée `P10`. Si le composant de contrôle jouant le rôle `Exécuteur` n'a pas démarré son exécution, rien ne se passe et le jeton est détruit (tir de `T10`). Si l'exécution a déjà démarré, la demande d'arrêt est prise en compte, si le composant rôle `RôleMaître` est bien celui qui avait initialement demandé le démarrage d'exécution (tir de `T9` et dépôt d'un jeton contenant les paramètres d'exécution dans la place de sortie `P15`). Si le composant rôle `RôleExécuteur` a déjà démarré l'exécution du composant de contrôle jouant le rôle `Exécuteur`, mais qu'un composant `RôleMaître` différent de celui ayant initialement demandé l'exécution fait une demande d'arrêt, alors cette demande d'arrêt n'est pas prise en compte et le jeton associé à cette demande est détruit (tir de `T8`). Lorsque le composant de contrôle renvoie la confirmation d'arrêt via un port `PAs6`, un jeton contenant ces paramètres est placé dans la place d'entrée `P16`. La transition `T11` est alors tirée si la demande d'arrêt a été faite (i.e. place `P12` marquée) et un jeton est déposé dans la place de sortie `P11`. Le jeton est alors émis vers la place `P8` (via la connexion entre les ports `PAs12` et `PAs9`). La transition `T4` de `RôleMaître` est alors tirée si le `RôleMaître` avait demandé une exécution (place `P5` marquée) et un jeton contenant les paramètres d'arrêt est placé dans la place de sortie `P3`. Cette version du descripteur de connecteur `ContrôleExécution` est simplifiée. Une version plus complète permettrait, par exemple, à un `RôleMaître` de gérer les refus de démarrage (afin que ces situations ne soient pas ignorées mais prises en compte), ce qui lui permettrait de retrouver un état cohérent suite à une telle situation. Notons que dans le cas où il y a plusieurs `RôleMaître`, le port `PAs12` du `RôleExécuteur` sélectionne le receveur du message en fonction de la donnée `Id` contenue en début de jeton. Des connecteurs `ContrôleExécution` peuvent donc être utilisés pour assembler plus de deux composants de contrôle.

9.3.4 Descripteur `NotificationEvénements`

Le deuxième descripteur de connecteur présenté est `NotificationEvénements` (cf. fig. 9.8). Un connecteur de ce type encapsule protocole d'abonnement/notification d'événements. Ce type de connecteur est utilisé à différents niveaux de l'architecture logicielle, en particulier entre les générateurs d'événements et les composants de contrôle interagissant avec eux. Le descripteur de connecteur `NotificationEvénements` est décrit à partir de deux descripteurs de composants rôles :

RôleNotifieur et RôleEcouteur. Le rôle Ecouteur est celui à travers lequel un composant de contrôle s'abonne à l'écoute d'événements notifiés par le composant de contrôle jouant le rôle Notifieur, et via lequel il reçoit une notification. Typiquement, un composant générateur d'événements joue le rôle Notifieur et des composants de contrôle tels que des *actions* et des *modes* jouent un rôle Ecouteur. Le générateur d'événements a la responsabilité de détecter l'occurence de l'événement, alors que le RôleNotifieur qu'il joue est chargé de notifier cette occurrence à tous les composants de contrôles jouant le rôle Ecouteur qui se sont abonnés auprès du RôleNotifieur.

Le descripteur de composant rôle RôleEcouteur possède trois descripteurs de ports asynchrones fournis PAs1, PAs2, PAs3 :

- PAs1 référence l'interface In abonner(Any anyEventCriterion). Il décrit un point de réception des demandes d'abonnements, provenant d'un composant de contrôle jouant le rôle Ecouteur.
- PAs2 référence l'interface In désabonner(Any anyEventCriterion). Il décrit un point de réception des demandes de désabonnement de l'écoute d'un événement, provenant d'un composant de contrôle jouant le rôle Ecouteur.
- PAs3 référence l'interface Out notification(anyEvent). Il décrit un point d'émission d'un événement détecté vers un composant de contrôle jouant le rôle Ecouteur.

L'interface de rôle Ecouteur, référencée par le descripteur de port fourni de Ecoute Evénements, contient ces trois interfaces asynchrones (cf. fig. 9.8). Le paramètre anyEvent permet de paramétrer le type d'événement émis par un composant rôle instance de RôleEcouteur. Le paramètre anyEventCriterion permet de paramétrer le type de critère à partir duquel l'événement est détecté.

Le descripteur de composant rôle RôleNotifieur possède trois descripteurs de ports asynchrones requis PAs4, PAs5 et PAs6 :

- PAs4 référence l'interface In démarrerDétection(anyEventCriterion). Il décrit un point d'émission de demandes de démarrage de la détection d'un événement vers un composant de contrôle jouant le rôle Notifieur.
- PAs5 référence l'interface In arrêterDétection(anyEventCriterion). Il décrit un point d'émission de demandes d'arrêt de la détection d'un événement vers un composant de contrôle jouant le rôle Notifieur.
- PAs6 référence l'interface Out notifierDétection(anyEventCriterion). Il décrit un point de réception des événements détectés provenant du composant de contrôle jouant le rôle Notifieur.

L'interface de rôle Notifieur, référencée par le descripteur de port requis de NotificationEvénements, contient ces trois interfaces asynchrones (cf. fig. 9.8). Dans le protocole défini par NotificationEvénements, un unique composant de contrôle joue le rôle Notifieur et un ou plusieurs composants de contrôle jouent le rôle Ecouteur.

Les comportements asynchrones des deux descripteur de composant rôles du connecteur NotificationEvénements sont présentés dans la figure 9.8. Lorsqu'un jeton arrive dans la place d'entrée P1, associée à un port de type PAs1, cela correspond à une demande d'abonnement du composant de contrôle jouant le rôle Ecouteur. La transition T1 ou la transition T5 est alors tirée. Le tir de T1 est effectué si le composant RôleEcouteur est déjà abonné à l'écoute de cet événement. Le tir de T5 est effectué si le composant rôle RôleEcouteur n'est pas déjà abonné à l'écoute de cet événement. Ce tir entraîne le dépôt d'un jeton dans la place de sortie P6, qui est associée à un port de type PAs6. Ce jeton contient l'identité du composant RôleEcouteur émetteur du message (Id), ainsi que l'EventCriterion qui permet de savoir quel critère permet de détecter l'événement. Le jeton est alors transmis vers l'unique port de descripteur PAs10 connecté avec chaque port de descripteur PAs7 : ce jeton est déposé dans la place d'entrée P9 associée. Ensuite, T7 ou T8 est tirée, T8 étant prioritaire vis à vis de T7. T8 est tirée si l'écoute de l'événement concerné (décrit par anyEventCriterion) est déjà

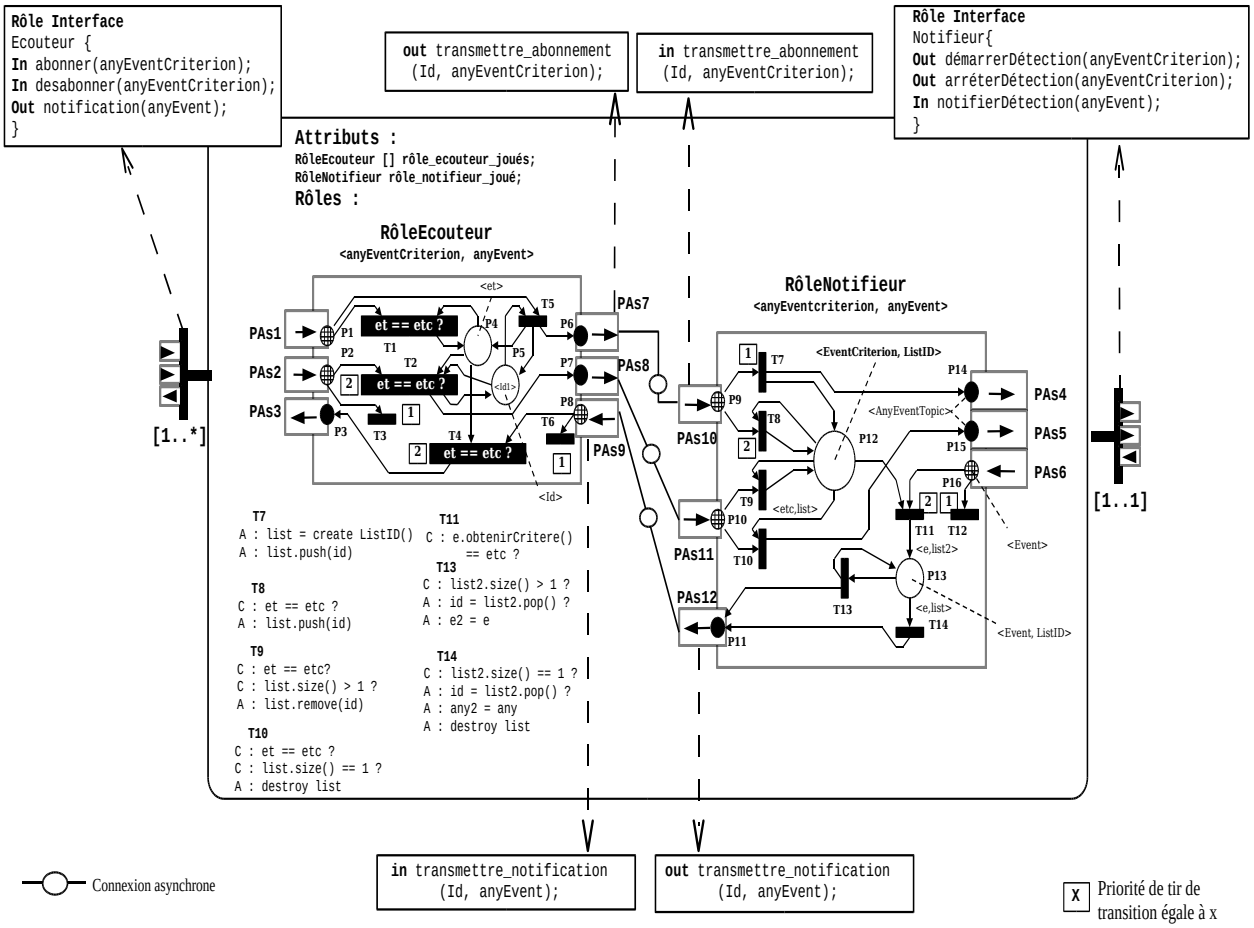


FIG. 9.8 – Représentation graphique du descripteur de connecteur `NotificationEvenements`

réalisée. Dans ce cas le composant rôle RôleNotifieur enregistre le nouvel écoutant de l'événement (il enregistre son Id) dans un liste d'écouteurs (listID). T7 est tirée lorsque l'écoute de l'événement considéré n'est pas réalisée au moment où la demande est faite, c'est à ce moment qu'est créée la liste des écouteurs d'un événement (correspondant donc à un critère particulier). Dans ce cas, un jeton est déposé dans la place P14 associée, qui correspond à l'unique port de descripteur PAs4, ce qui entraîne le démarrage du mécanisme de détection de l'événement considéré.

Le composant de contrôle jouant le rôle Ecouteur peut se désabonner de l'écoute d'un événement. Dans ce cas, un jeton est déposé dans la place d'entrée P2 correspondant à un port de descripteur PAs2. Si l'écoute de l'événement est réalisée (place P4 marquée), la transition T2 est tirée et un jeton déposé dans la place P7 associée au port de descripteur PAs8. Il est alors transmis au port de descripteur PAs11, connecté au port de descripteur PAs8. Ce jeton est ensuite déposé dans la place d'entrée P10. Là, soit T9 est tirée, soit T10. Dans le cas où T9 est tirée, l'identificateur du composant rôle RôleEcouteur est retiré de la liste des écouteurs de l'événement considéré. Dans le cas où T10 est tirée, la liste des écouteurs de l'événement considéré est détruite car elle a été entièrement vidée de ses écoutants. Un jeton est alors déposé dans la place de sortie P15 qui est associée au port de descripteur PAs5 : cela entraîne l'arrêt de la détection d'événements pour le composant de contrôle jouant le rôle Notifieur. Lorsqu'un jeton arrive dans la place d'entrée P16, associée à l'unique port de descripteur PAs6, cela correspond à une détection d'un événement qui doit être notifié (dépôt d'un jeton contenant l'événement dans la place P16). Dans le cas où l'événement n'était pas écouté, la transition T12 est tirée et le jeton détruit. T11 est tirée prioritairement si l'évenement considéré était bien écouté par au moins un composant rôle RôleEcouteur : chaque écouteur enregistré dans la liste des écouteurs est alors notifié de l'événement (tir de T13) et la liste en elle même est détruite au moment où le dernier écouteur est notifié (tir de T14). Cela entraîne le dépôt d'un ou plusieurs jetons (un par écouteur) dans la place P11, puis la transmission de chaque jeton dans la place P8 de chaque composant rôle RôleEcouteur concerné (il y a sélection du recéveur de la notification en fonction des Id enregistrés). La transition T6 est tirée si l'événement n'était pas écouté par le composant rôle RôleEcouteur concerné (ce qui correspondrait alors à une erreur de destinataire). Sinon, la transition T4 est tirée, ce qui entraîne le dépôt du jeton contenant l'événement dans la place de sortie P3, associée au port de descripteur PAS3. Chaque composant de contrôle jouant le rôle Ecouteur reçoit alors l'événement considéré.

Les connecteurs utilisés au sein de l'architecture ayant été modélisés, les sections suivantes présentent les différents composants "métier" utilisés au sein de la description de l'architecture du contrôleur du robot mobile.

9.3.5 Composants de représentation

Différents composants de représentation sont utilisés dans cette modélisation. La figure 9.10 en présente quelques uns. Ces composants de représentation représentent les connaissances manipulées au sein du contrôleur de robot mobile. Par exemple le descripteur de composant de représentation VéhiculeNon-holonome représente la connaissance complète sur le véhicule du robot. Il représente le type de Véhicule considéré, aussi bien ses propriétés physiques (e.g. éléments matériels qui le constituent) que géométriques. La figure 9.9 présente certaines des propriétés géométriques intéressantes du véhicule.

Le service IPositionVéhicule (cf. fig. 9.10) permet de paramétrer les valeurs "dynamiques" du véhicule (vitesse et direction) et de consulter la position courante du véhicule. Le descripteur de composant VéhiculeNon-HolonomeCeintureUltrasons spécialise le descripteur précédent et possède en plus un descripteur de port fourni permettant d'accéder au service ICeintureUltrasons. Ce service permet de tester la présence d'obstacle (à partir des mesures des capteurs à ultrasons) ou de calculer l'enveloppe virtuelle du véhicule.

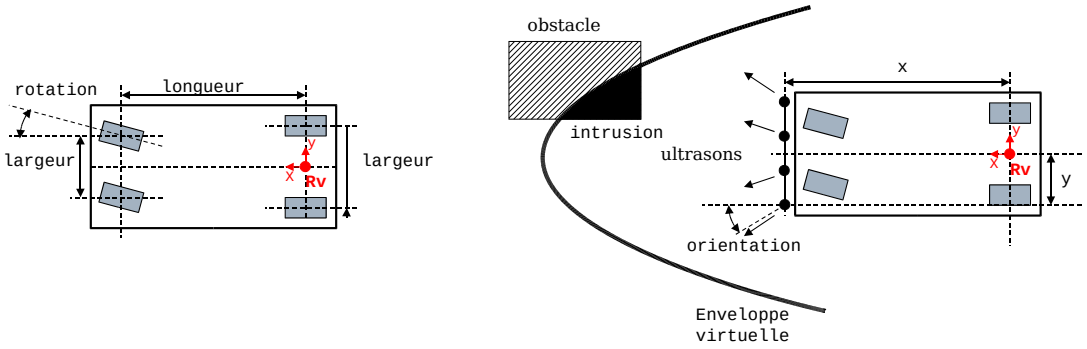


FIG. 9.9 – Propriétés géométriques du véhicule et son enveloppe virtuelle

Le descripteur de composant `LoiCalculCommandePositionVéhicule` représente le type de la loi mathématique permettant de calculer les données de commande des moteurs du véhicule en fonction des données des capteurs du véhicule et d'une consigne. Il fournit un service `IloiPositionVéhicule` qui donne la possibilité de : fixer le jeu de gains (proportionnel, intégral, dérivé) pour paramétrer la dynamique de la loi, fixer la consigne de position à atteindre et de réaliser le calcul de la commande, etc. Il requiert un service `IPositionVéhicule` ainsi qu'un service `IPropriétéVéhiculeNon-Holonaume`, via lequels il calcule la donnée de commande en fonction des propriétés physiques, géométriques et cinématiques du véhicule.

Le descripteur de composant `EvénementObstacleProche` représente le type d'événement relatif à la présence d'un obstacle proche du robot. Il possède un attribut `eovt` lui permettant de paramétrer le critère de détection du phénomène (e.g. la distance de proximité d'un obstacle). Il contient un descripteur de port fourni qui référence l'interface déclarant le service `IEvénementObstacleProche`. Ce service est décomposé en sous services permettant de : tester l'occurence de l'événement, de consulter la dernière date d'occurrence de l'événement, d'obtenir le critère de détection de l'événement, d'évaluer la distance à la position de l'obstacle. Ce composant utilise le service `ICeintureUltrasons` via son port requis, afin de consulter les valeurs des données ultrasons à partir desquelles il effectuera les calculs liés à la détection d'événements.

Ces composants de représentation sont utilisés dans différents composants de contrôle, présentés par la suite.

9.3.6 Commande Véhicule En Position

Le composant de contrôle `CommandeVéhiculePosition` applique périodiquement une loi de commande qui permet d'asservir le véhicule en fonction de consignes de position. Il envoie périodiquement des consignes de direction et de vitesse au `ControleurE/SMicrocontrôleur` qui se charge de les appliquer (le microcontrôleur effectuant un asservissement en vitesse). Le composant `CommandeVéhiculePosition` possède deux ports de contrôle fournis et deux ports de contrôle requis. Le port de contrôle fourni `P46`, typé par l'interface `ICommanderVehiculePosition` donne l'accès au service de commande du véhicule, qui permet de démarrer et arrêter l'application de la loi de commande en position. Le port de contrôle fourni `P45`, typé par l'interface `IParamétrerGains` donne l'accès au service de paramétrage des gains de la loi de commande appliquée en modifiant la valeur de ses gains.

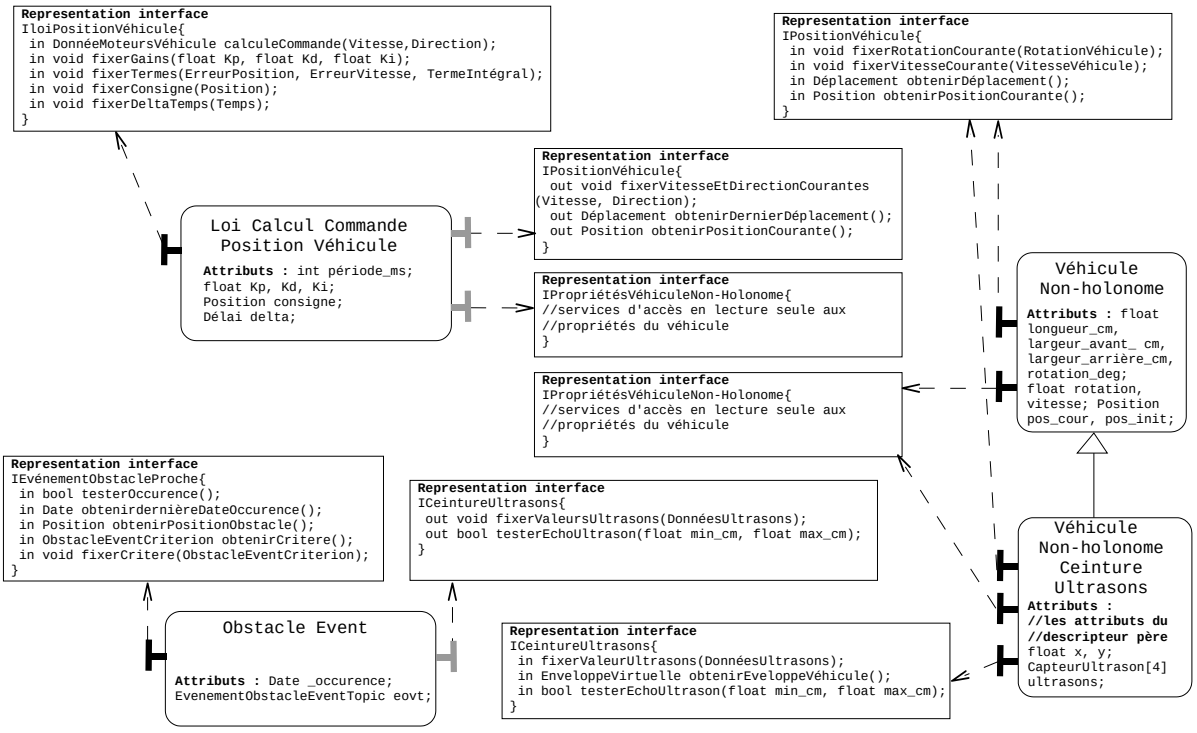


FIG. 9.10 – Composants de représentation `Véhicule, événement et loi`

Le port de contrôle requis `P53` typé par l'interface `IPerceptionPositionVéhicule`, permet au composant d'utiliser un service via lequel il obtient répétitivement la position courante du véhicule. Le port requis `P52` lui permet d'accéder au service `IAccèsMoteurDirectionEtVitesseVéhicule` via lequel il peut agir (indirectement, via le composant `contrôleurE/SMicrocontrôleur` dans notre cas) sur le moteur (vitesse) et le servomoteur (direction) du véhicule.

Le composant `CommandeVéhiculePosition` (cf. fig. 9.11) a pour attributs :
- `loi_appliquée` est un composant de représentation de type `LoiCalculCommandePositionVéhicule`.
- `véhicule_contrôlé` est un composant de représentation de type `VéhiculeNon-HolonomeCeintureUltrasons`.
- `période` est un entier, représentant la période d'application de la loi en millisecondes.
- `relation_loi_vehicule_position` et `relation_loi_vehicule_propriétés` sont les deux connexions entre `loi_appliquée` et `véhicule_contrôlé`.

A l'initialisation de `CommandeVéhiculePosition`, `loi_appliquée` et `véhicule_contrôlé` sont instanciés et assemblés via les deux connexions également créées. les composants référencés par `loi_appliquée` et `véhicule_contrôlé`) sont ensuite placés dans un jeton, qui est lui-même ajouté au marquage de la place `P16`.

Le comportement asynchrone de `CommandeVéhiculePosition` est décrit par le RdPOC présenté dans la figure 9.11. L'activité de `CommandeVéhiculePosition` est démarrée lors de l'arrivée, dans la place d'entrée `P1`, d'une consigne de position du véhicule, à atteindre. Le tir de la transition `T1` correspond au démarrage réel du composant tandis que le tir de la transition `T2` correspond à une mise à jour de la consigne de position à atteindre. Le tir de `T1` entraîne :

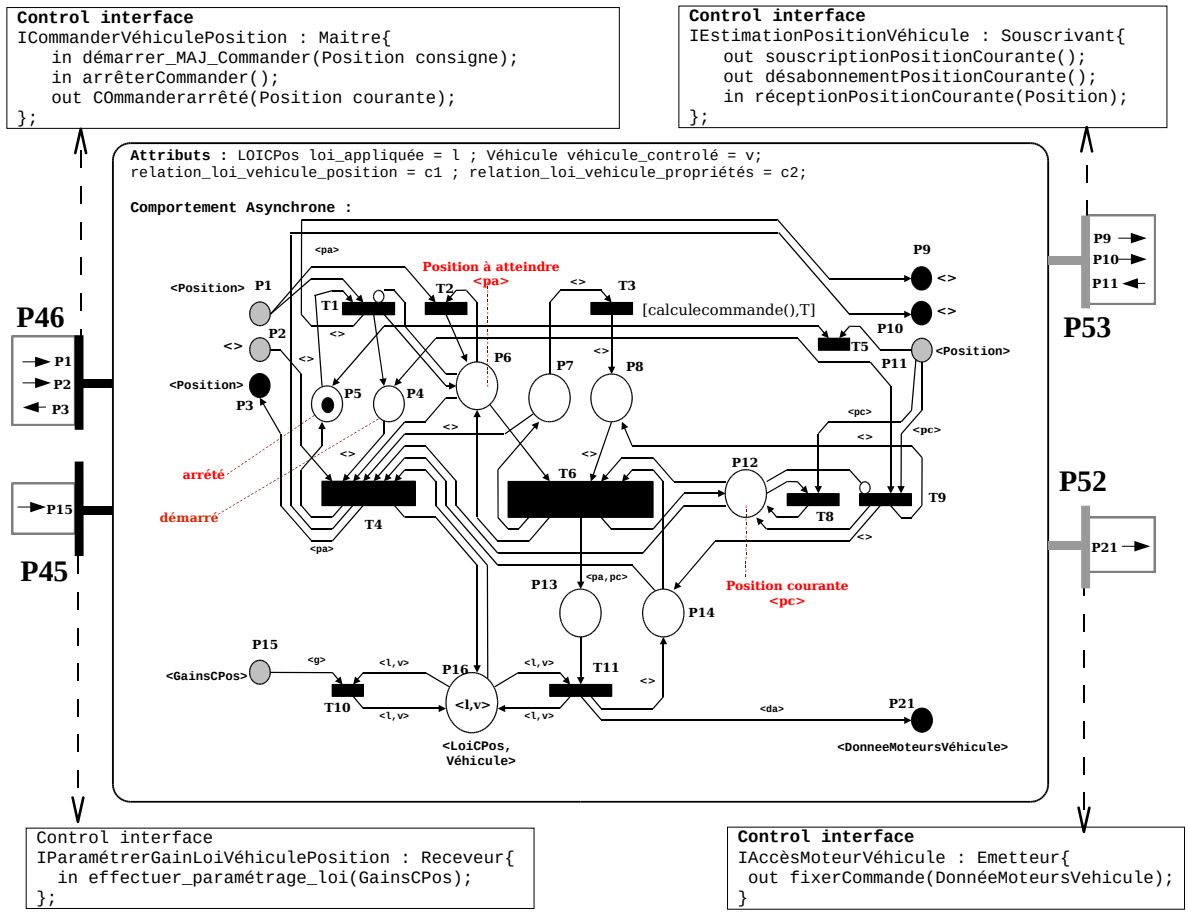


FIG. 9.11 – Le composant de contrôle `CommandeVéhiculePosition`

- le dépôt d'un jeton dans la place P9, ce qui équivaut à une demande d'écoute périodique de la position du véhicule (émission d'un message souscriptionPositionCourante()).
- le retrait du jeton de la place P5 et son dépôt dans la place P4, ce qui signifie que le composant vient de passer de l'état *arrêté* à l'état *démarré*. T2 devient alors franchissable si un jeton arrive dans P1.

Les transitions T5, T8 et T9 permettent de gérer l'arrivée périodique de jetons dans la place d'entrée P11 afin de mettre à jour la position courante, contenue dans la place P12. Une fois que CommandeVéhiculePosition est dans l'état *démarré* et qu'il a reçu une première donnée de position du véhicule, son cycle d'asservissement démarre. Ce cycle d'asservissement est exprimé par les transitions T11, T6 et T3. La transition T6 est tirée, ce qui entraîne la copie des positions courante et à atteindre dans un jeton, qui est alors déposé dans la place P13. La transition T11 est ensuite tirée, consommant le jeton de la place P13 et utilisant le jeton contenant la loi_appliquée et le véhicule_controlé. Pendant le tir de T11, les positions courante et à atteindre sont passées en paramètre du service calculeCommande de la loi_appliquée, qui calcule la réaction à apporter (c'est-à-dire les données de commande au moteur et au servomoteur du véhicule). Le jeton, contenant la DonnéeMoteursVéhicule générée par l'appel de calculeCommande, est placé dans la place de sortie P21, ce qui correspond à une demande d'actualisation des commandes appliquées aux moteurs du véhicule. Après le tir de T11, la place P14 est à nouveau marquée ce qui veut dire qu'un nouveau cycle peut reprendre. Lorsque T6 est tirée, la place P7 est marquée, la transition T3 est alors sensibilisée. Elle ne sera tirée que lorsque le temps renvoyé par calculerTemps est écoulé (ce temps étant calculé à partir de l'attribut période). Au bout de ce temps calculerTemps, T3 est tirée et un jeton est placé dans P8 ; un nouveau cycle d'asservissement peut donc être exécuté. Lorsque la place P2 est marquée, cela correspond à l'arrêt de l'asservissement, le RdPOC reprend alors son état initial.

9.3.7 Générateur Evénements Obstacles Proches Véhicule

Le composant GénérateurEvénementsObstaclesProchesVéhicule permet, dans le cadre de la ressource Mobile, de détecter des événements liés à l'approche ou à l'éloignement d'obstacles pendant l'évolution du véhicule dans l'environnement. Il est présenté dans la figure 9.12. Un composant de contrôle GénérateurEvénementsObstaclesProchesVéhicule permet de générer un événement s'il détecte (ou ne détecte plus) d'obstacles proches du véhicule. Il possède un port de contrôle qui fournit le service IDétectionObstacleVéhicule (service de détection d'obstacles), qui permet à des composants de s'abonner, de se désabonner à la notification d'événements liés à la détection d'obstacles. Les composants de représentation ObstacleEventCriterion permettent de stipuler le type d'événements possible lors de cette détection ("obstacle proche" ou "plus d'obstacle proche") ainsi que les paramètres liés à ce type d'événement (distance entre le véhicule et l'obstacle définissant la proximité seuil). Il notifie l'occurence d'un événement via l'envoi d'un composant de représentation de type ObstacleEvent qui contient les paramètres liés à cet événement (type d'événement, éloignement, date d'occurence). GénérateurEvénementsObstaclesProchesVéhicule possède également un port de contrôle requis typé par l'interface IAccèsMesureUltrasons, via lequel il récupère les mesures des capteurs à ultrasons, publiées par le ContrôleurE/SMicrocontrôleur dans l'architecture du Mobile.

Le comportement asynchrone d'un composant GénérateurEvénementsObstaclesProchesVéhicule est présenté dans la figure 9.12. Il est essentiellement basé sur l'utilisation du service IAccèsMesureUltrasons grâce auquel il reçoit périodiquement les mesures des faisceaux ultrasons. Le RdPOC décrit la façon dont la détection des ObstacleEvent est réalisée à partir des mesures ultrasons générées par le ContrôleurE/SMicrocontrôleur. Le tir de la transition T1 correspond à une demande de démarrage de la détection d'un obstacle en fonction d'un ObstacleEventCriterion (qui définit en particulier la distance à l'obstacle). Elle entraîne la création d'un composant de représentation ObstacleEvent et sa connexion avec le composant de représentation VéhiculeNonHolonomeCeinture-

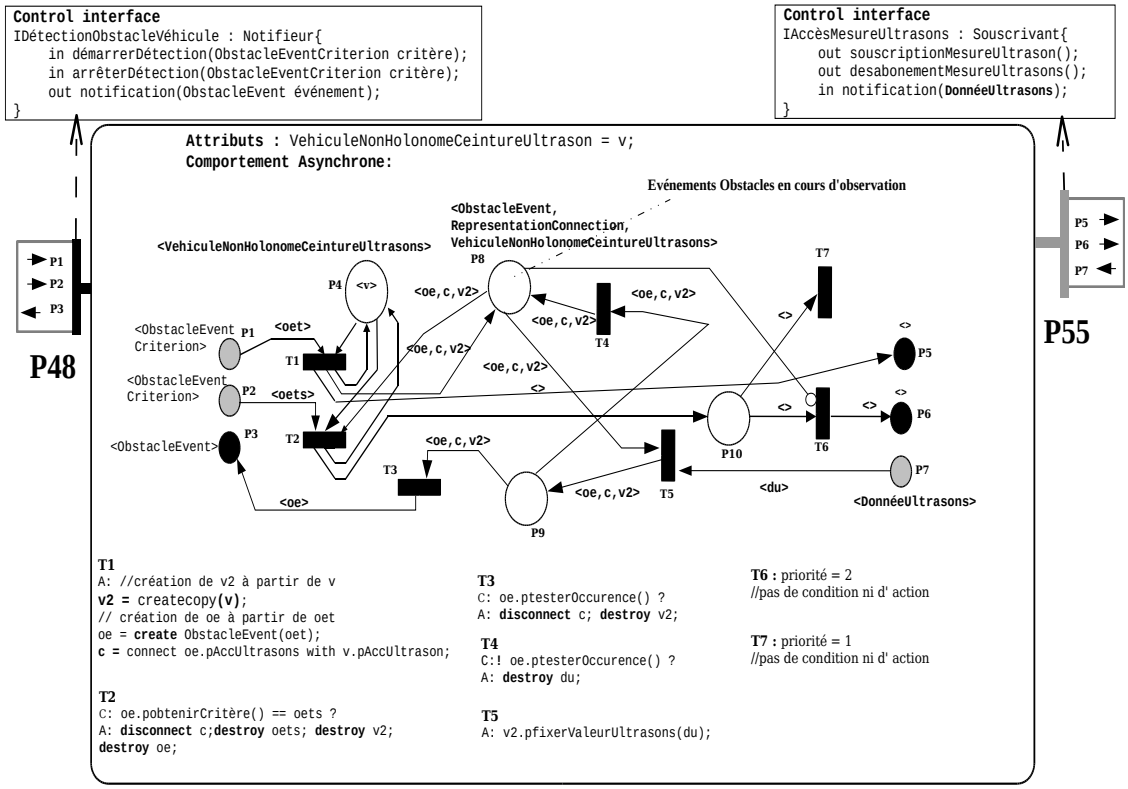


FIG. 9.12 – Le composant de contrôle `GénérateurEvénementsObstacleProcheVéhicule`

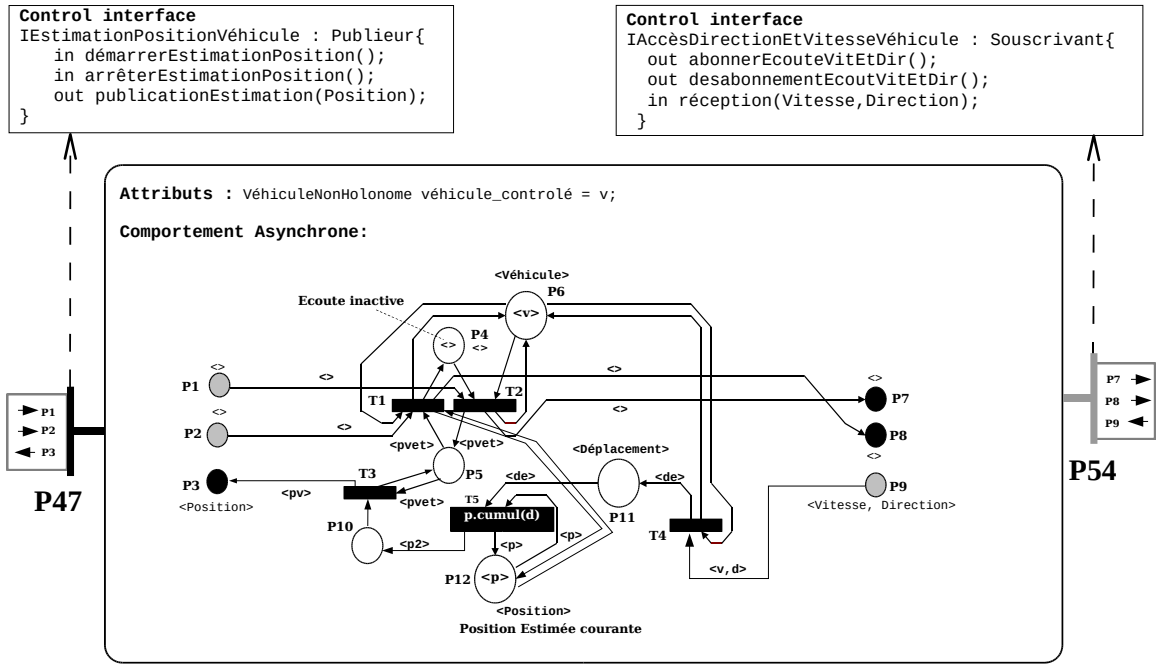


FIG. 9.13 – Le composant de contrôle `PerceptionPositionVéhicule`

`Ultrasons`, qui intègre les propriétés du capteur à ultrasons utilisé pour la détection. Le jeton correspondant à ce couple est alors placé dans la place `P8`. Le tir de `T1` amène également à la souscription du `GénérateurEvénementsObstaclesProchesVéhicule` à l'écoute périodique de données de type `DonnéesUltrasons` (propagation du jeton vers la place `P5`). Le tir de la transition `T2` correspond à l'arrêt de la détection d'un `ObstacleEvent`. Cela peut déboucher sur un désabonnement de la notification à des `DonnéesUltrason` (tir de `T6` et dépôt d'un jeton dans la place de sortie `P6`), dans le cas où il n'y a plus aucun `EventObstacle` en cours de détection (place `P8` vide). Dans le cas contraire, la transition `T8` est tirée et le jeton supprimé. Lorsqu'un jeton arrive dans la place d'entrée `P7`, cela correspond à l'arrivée d'une nouvelle `DonnéeUltrasons`. La transition `T5` est alors tirée, ce qui représente la mise à jour des mesures ultrason au sein du `VéhiculeNonHolonomeCeintureUltrasons`. Ensuite, le tir de la transition `T3` correspond à la notification de l'`ObstacleEvent` dont l'occurence vient d'être détectée : il s'ensuit le dépôt d'un jeton dans la place de sortie `P3`. Le tir de `T4` a lieu dans le cas contraire.

9.3.8 Perception Position Véhicule

Parmi les composants de contrôle *perception* de notre architecture, nous présentons le composant `PerceptionPositionVéhicule` (cf. fig. 9.13). A partir des données issues des capteurs placés sur le véhicule (vitesse et direction), ce composant de contrôle est chargé d'estimer la position relative du véhicule dans l'environnement, dans un repère initial fixé au démarrage de l'activité du composant. Ce composant possède un port de contrôle fourni typé par l'interface `IEstimationPositionVéhicule` qui permet publie la position courante estimée du véhicule. `PerceptionPositionVéhicule` possède un port de contrôle requis, typé par l'interface `IAccèsDirectionEtVitesseVéhicule` qui lui permet d'être périodiquement alerté par le `contrôleurE/Smicrocontrôleur` des nouvelles valeurs de direction et de vitesse du véhicule.

Le comportement asynchrone de PerceptionPositionVéhicule décrit le mécanisme d'estimation de la position relative du véhicule. L'estimation démarre lorsque la transition T2 est tirée, après arrivée d'un jeton dans la place P1. A partir du moment où T2 est tirée, plus aucune autre demande d'estimation de la position ne sera satisfaite, ce qui est exprimé par le démarquage de la place P4. Un jeton est posé dans la place P7, ce qui correspond à un abonnement à la réception périodique des vitesse et direction courantes du véhicule. Une fois l'estimation démarrée, la place P5 est marquée, le composant PerceptionPositionVéhicule va avoir un comportement cyclique, car influencé par la publication périodique de données Vitesse et Direction (arrivée d'un jeton dans la place d'entrée P9). A chaque arrivée d'une nouvelle donnée de position et de vitesse, la transition T4 est tirée, afin de convertir la donnée de vitesse et direction, en une donnée caractérisant le déplacement réalisé par le véhicule. Le composant de représentation de type Déplacement est généré en fonction des caractéristiques physiques du véhicule (appel du service du composant de représentation VéhiculeNonHolonome) ; il est contenu dans un jeton déposé dans la place P11. Lorsque la transition T5 est tirée, la nouvelle position estimée est calculée par cumul du nouveau déplacement. La position courante est mise à jour par dépôt d'un jeton dans la place P12. Notons qu'initialement, la position contenue dans P12 est fixée à la position initiale d'un repère relatif. Ce repère relatif correspond à celui du véhicule au moment du démarrage du service IEstimationPositionVéhicule. Une copie de la position courante estimée est faite (la variable p2 est créée), puis placée dans un jeton déposé dans la place P10. La transition est alors tirée et un composant de représentation de type Position est créé puis émis vers la place de sortie P3. Ceci correspond à une notification d'une nouvelle position estimée du véhicule. L'arrêt de l'estimation de la position est déclenché par l'arrivée d'un jeton dans la place d'entrée P2, ce qui entraîne le tir de la transition T1 : la place P5 étant démarquée, l'activité cyclique de PerceptionPositionVéhicule est interrompue ; la position courante du véhicule est réinitialisée (remise à zéro). Le tir de T1 amène le dépôt d'un jeton dans la place P8, ce qui correspond au désabonnement de l'écoute périodique des vitesse et direction courantes du véhicule. Cela amène également le dépôt d'un jeton dans la place P4, ce qui traduit le fait que le composant mémorise l'arrêt de son activité d'estimation de la position du véhicule.

9.3.9 Action Opérateur Pilote Véhicule

Le composant de contrôle ActionOpérateurPiloteVéhicule permet à un opérateur de piloter le véhicule à distance et en "temps réel", via un lien de téléopération direct [8]. Pour cela, il intègre un serveur de téléopération, afin de gérer les retards de communication variables, entre le robot et la plate-forme opérateur. Il se base sur le calcul des retards de communication (RTT) afin de connaître la qualité des communications avec l'opérateur et intègre un mécanisme de stockage des consignes provenant de l'opérateur dans une pile. A partir du RTT calculé, il choisit la dynamique appropriée pour l'asservissement du véhicule. Il utilise un composant de contrôle fournissant le service de commande en position du véhicule (fourni, dans notre architecture, par le composant de contrôle CommandeVéhiculePosition) afin de réaliser cet asservissement. La dynamique d'asservissement calculée par ActionOpérateurPiloteVéhicule, détermine la vitesse à laquelle sont dépilées les consignes et la façon dont est configurée l'exécution de CommandeVéhiculeEnPosition (modification des termes de la loi de commande).

La figure 9.14 présente ce composant de contrôle, qui possède deux ports de contrôle fournis, l'un (P32) typé par l'interface IActionOpérateurPiloterVéhicule et l'autre (P33) typé par l'interface IDétectionretardDeCommunicationLong. P32 donne accès au service permettant d'établir le lien de téléopération direct entre l'opérateur et le véhicule. P33 permet de notifier le fait que le RTT est trop important et met donc en péril la téléopération. ActionOpérateurPiloteVéhicule possède un port de contrôle requis, typé par l'interface IEstimationPositionVéhicule, via lequel il peut connaître la position estimée courante du véhicule. Il possède également un port requis typé par l'interface ICommanderVéhiculePosition, via lequel il peut contrôler l'asservissement du véhicule.

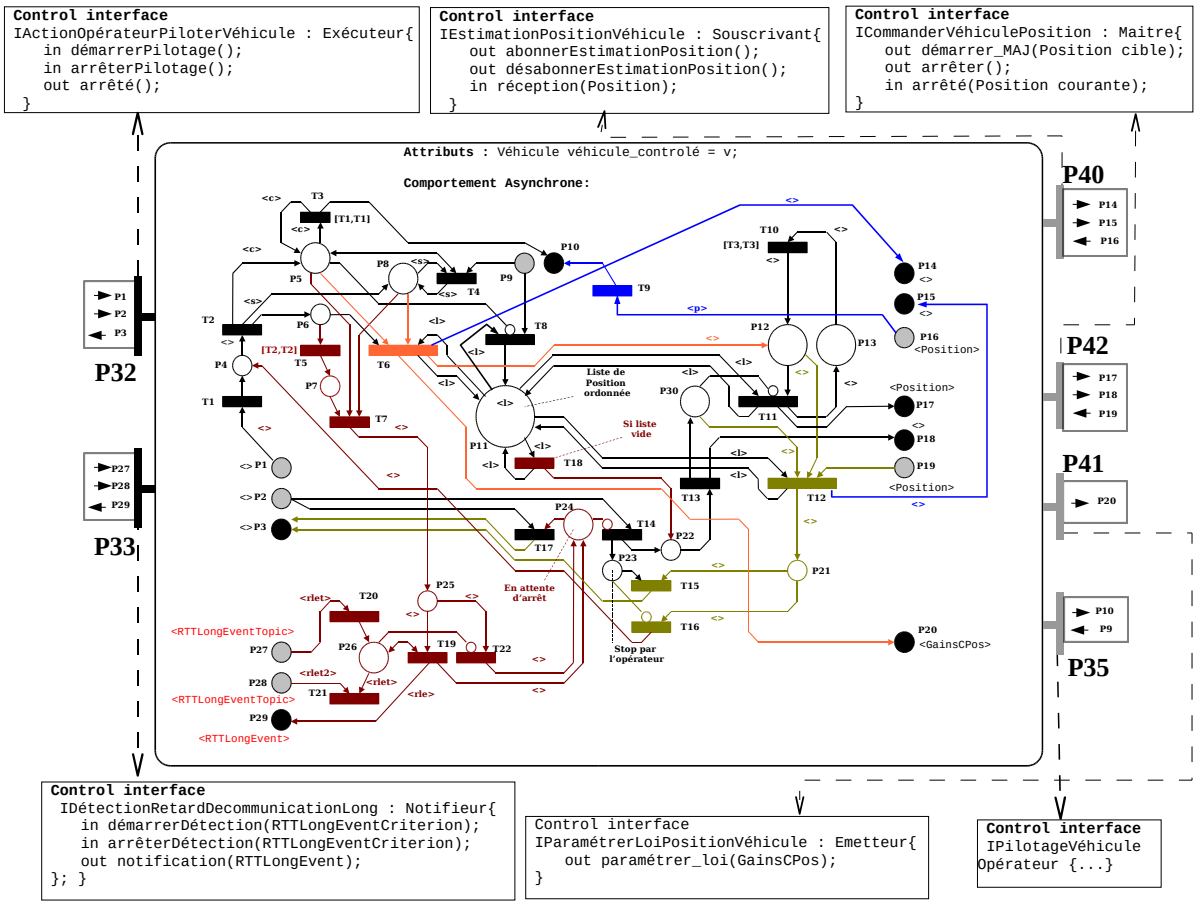


FIG. 9.14 – Le Composant de contrôle `ActionOpérateurPiloteVéhicule`

Enfin, il possède un port requis, typé par l'interface `IPilotageVéhiculeParOpérateur`, via lequel il demande explicitement à l'opérateur de lui fournir des consignes.

Le comportement asynchrone est présenté dans la figure 9.14. Lorsqu'un jeton arrive dans la place d'entrée `P1`, le composant `IActionOpérateurPiloterVéhicule` démarre son activité de serveur de téléopération. La transition `T1` est tirée et un jeton placé dans la place `P4`, ce qui correspond au lancement de la phase d'initialisation. Cette phase d'initialisation débute par le tir de `T2`, qui génère un jeton contenant un composant de représentation `compteur` dont la valeur est intialisée à 100, et le dépôt de ce jeton dans la place `P5`. Un jeton contenant un composant de représentation `Statistique` est placé dans la place `P8`. Enfin, un jeton est déposé dans la place `P6`. `T5` possède une fenêtre temporelle dont le temps au plus tôt détermine la durée maximale de la phase d'initialisation, et permet au composant de se désengager de cette phase si elle dépasse ce temps. La transition `T3` est tirée périodiquement autant de fois que la valeur initiale du `compteur c` le précise (100 par défaut donc) et génère un jeton déposé dans la place de sortie `P10`. Ceci correspond à l'émission des trames de test vers la plate-forme opérateur. Les réponses de la plate-forme opérateur arrivent sous la forme de trames de test dans la place `P9`. A chaque réponse la transition `T4` est tirée afin de mettre à jour le composant de représentation `Statistique s`, qui permet de déterminer la moyenne des RTT et le RTT maximal. Lorsque le `compteur c` a atteint la valeur zéro, la transition `T3` n'est plus franchissable et la transition `T6` le devient. Lorsque `T6` est tirée le jeton `<s>`, contenant la statistique de l'échange, est consommé. A partir de ces statistiques, les termes de la `loiCPos` sont calculés afin de donner au

véhicule une dynamique qui soit adaptée à la qualité du lien de téléopération. Le jeton correspondant est placé dans la place de sortie P20. Le composant de représentation Statistique permet également de configurer la taille de la liste de positions l qui sert de pile FIFO pour mémoriser les consignes fournies par l'opérateur. Enfin, le tir de T6 engendre la souscription à l'écoute de la position courante du véhicule (dépôt d'un jeton dans la place de sortie P14). L'interruption de la phase d'initialisation est représentée par la transition T7, elle est présentée plus loin.

A la fin de l'initialisation, l'opérateur envoie des consignes de déplacement qui sont déposées dans la place d'entrée P9. Elle sont alors consommées par la transition T8, qui va ajouter cette consigne dans la liste des consignes contenue dans la place P11. L'ordre des consignes n'étant pas assuré (communication via UDP), la liste de consignes est un composant de représentation qui réordonne les consignes en fonction de l'information contenue dans la trame de consigne envoyée par l'opérateur. La position estimée est reçue dans la place d'entrée P15, et la transition T9 est tirée : une trame de position est envoyée à la plate-forme opérateur, via le dépôt d'un jeton dans la place de sortie P10. Cela permet à la plate-forme opérateur de recevoir périodiquement un retour de la position du véhicule, ce qui permet à l'opérateur de se tenir informé des déplacements réalisés.

Après l'initialisation, la liste des consignes de position commence à se remplir avec les consignes provenant de l'opérateur. Cette liste est un tampon qui permet d'absorber la fluctuation du RTT à hauteur du RTT maximum. Dès que cette liste contient le nombre de consignes requis (calculé en fonction du RTT maximum), T11 est tirée. Un jeton est alors déposé dans la place de sortie P17, ce qui correspond à une demande de démarrage de l'activité du composant CommandeVéhiculePosition, ou à une mise à jour de sa consigne de position. Les transitions T10 et T11 permettent de mettre en place une boucle périodique de mise à jour des consignes de position. Si la liste des consignes est vide, les transitions T18 et T13 sont tirées successivement : cela entraîne l'arrêt des mises à jour périodiques de consigne (marquage de la place P30) et l'arrêt d'application de la commande (dépôt d'un jeton dans la place de sortie P18). Lorsque la confirmation d'arrêt de l'activité du composant commande arrive dans la place P19, la transition T12 est tirée, ce qui vide la liste de consignes. Ce tir de T12 entraîne une demande de désabonnement à l'écoute de la position du véhicule (dépôt d'un jeton dans P15) et au marquage de P21. Par la suite, la transition T15 est tirée dans le cas où l'arrêt de la téléopération a été explicitement demandé (place P23 marquée). Sinon, la transition T16 est tirée, ce qui correspond à une reprise de la phase d'initialisation (dépôt d'un jeton dans la place P4) pour redimensionner la liste et réadapter la dynamique du véhicule à la qualité des communications entre le robot et la plate-forme opérateur. L'arrêt de la téléopération a lieu lorsqu'un jeton est déposé dans la place P2. Si le composant est en attente d'arrêt (place P24 marquée), la transition T17 est tirée et le jeton de confirmation d'arrêt est déposé dans la place P3. Sinon, la transition T14 est tirée, ce qui correspond à une demande explicite d'arrêt de la téléopération (dépôt d'un jeton dans la place P23) et à une demande d'arrêt de l'asservissement (dépôt d'un jeton en P22).

La phase d'initialisation en elle-même peut échouer, dans le sens où elle prend trop de temps. La transition T5 est tirée lorsque le délai imparti à l'initialisation est écoulé, la place P6 est démarquée et un jeton est déposé dans la place P7. La transition T7 est alors tirée (destruction des composants de représentation compteur et Statistique) et un jeton est déposé dans la place P25. Cet échec de l'initialisation peut donner naissance à un événement, si le mécanisme de détection des événements RTTLongEvent (représenté par les transition T19, T20 et T21) a été activé (par l'abonnement d'au moins un composant). Le tir de T19 a lieu si le mécanisme de détection est activé, ce qui entraîne le dépôt d'un jeton RTTLongEvent dans la place de sortie P29. Sinon, la transition T22 est tirée. Dans ces deux cas, un jeton est déposé dans la place P24, ce qui correspond au fait que le composant ActionOpérateurPiloteVéhicule est en attente d'arrêt.

Cet exemple est intéressant dans le sens où il montre la complexité que peut atteindre un comportement de composant de contrôle. Il l'est également car il montre qu'il est parfois difficile d'abstraire des composants de représentation représentant de "vraies" connaissances sur le "monde physique".

Il serait interressant de décrire le "lien de télé-opération" sous la forme d'un composant de représentation (son état étant l'état estimé du lien et ses opérations permettant d'évaluer la qualité du lien).

9.3.10 Mode Véhicule Téléopéré

Le composant de contrôle `ModeVéhiculeTéléopéré` (cf. fig. 9.15) permet d'engager le véhicule dans une relation de téléopération avec un opérateur. Il contrôle l'exécution du composant de contrôle `ActionOpérateurPiloteVéhicule` afin de donner à l'opérateur la responsabilité du pilotage du véhicule. Il s'enregistre auprès d'un `GénérateurEvénementsObstacleVéhicule` afin que celui-ci le notifie de la proximité d'obstacles. `ModeVéhiculeTéléopéré` possède un port fourni (`P21`) typé par l'interface `IModeTéléopérationVéhicule` via lequel la relation de téléopération peut-être contrôlée. Il possède un port fourni (`P22`) typé par l'interface `IDétectionTéléopérationImpossible` qui permet de notifier l'impossibilité de téléopérer le véhicule. `ModeVéhiculeTéléopéré` possède un port requis, qui référence l'interface `IActionOpérateurPiloterVéhicule`, via lequel il contrôle l'exécution du composant qui prend en charge le pilotage du véhicule. Il possède un port requis typé par l'interface `IDétectionObstacleVéhicule`, via lequel il s'enregistre à la notification d'événements lui signalant un obstacle proche du véhicule. Il possède un port requis typé par l'interface `IDétectionEvénementsWiFi` qui lui permet d'être informé lorsque le signal Wi-Fi est trop faible pour assurer une téléopération directe. Via ses ports de contrôle requis, il possède les informations nécessaires à un désengagement éventuel d'une relation de téléopération, lorsque un obstacle est considéré comme trop proche du véhicule, que le RTT est trop grand ou lorsque la liaison Wi-Fi est de trop mauvaise qualité.

Le comportement asynchrone du `ModeVéhiculeTéléopéré` est présenté dans la figure 9.15. Lorsqu'un jeton arrive dans la place `P1`, la transition `T1` est tirée : la place `P5` est démarquée et la place `P4` marquée ce qui correspond au passage du composant dans un état de *télé-opération active*. Les places de sortie `P11`, `P14` et `P24` sont alors marquées, ce qui correspond à l'enregistrement du composant auprès des services de notification d'événements, respectivement liés au retard de communication, à la proximité d'obstacle et au niveau de réception Wifi. La place de sortie `P8` est également marquée, ce qui correspond au démarrage effectif de la télé-opération et donc, dans l'architecture présentée, au démarrage de l'activité de l'action `ActionOpérateurPiloteVéhicule`. Lorsque ce composant notifie le `ModeVéhiculeTéléopéré` d'un événement `RTTLongEvent` un jeton est déposé dans la place `P13`. La transition `T7` est tirée, les places `P15` et `P25` sont marquées, ce qui correspond à un désabonnement à la notification d'événements de types respectivement `ObstacleEvent` et `NiveauRéceptionWifiFaibleEvent`. Le jeton contenant `RTTlongEvent` est déposé dans `P17`. Lorsque le composant est notifié d'un événement `ObstacleEvent`, un jeton est déposé dans la place `P16`. La transition `T8` est alors tirée, les places de sortie `P12` et `P25` sont alors marquées, ce qui correspond au désabonnement de la notification d'événements de types respectivement `RTTLongEvent` et `NiveauRéceptionWifiFaibleEvent`. Le jeton contenant l'événement `ObstacleEvent` reçu est placé dans la place `P17`. Lorsque le composant est notifié d'un événement `NiveauRéceptionWifiFaibleEvent`, un jeton est déposé dans la place `P26`. La transition `T13` est alors tirée, les places de sortie `P12` et `P15` sont alors marquées, ce qui correspond au désabonnement de la notification d'événements de types respectivement `RTTLong` et `ObstacleProcheVéhicule`. Le jeton contenant l'événement `NiveauRéceptionWifiFaible` reçu est placé dans la place `P17`. Pour résumer, chaque fois que le composant reçoit un événement de nature à remettre en cause la téléopération, il se désabonne à l'écoute de tous les autres événements et engage une procédure de désactivation automatique de son activité (qui débute par le dépôt d'un jeton dans la place `P17`). La transition `T6` est tirée : un jeton banalisé est placé dans la place `P18` (passage dans un état de téléopération impossible) ; un jeton contenant l'événement déclencheur est placé dans la place `P19` (possibilité d'émission

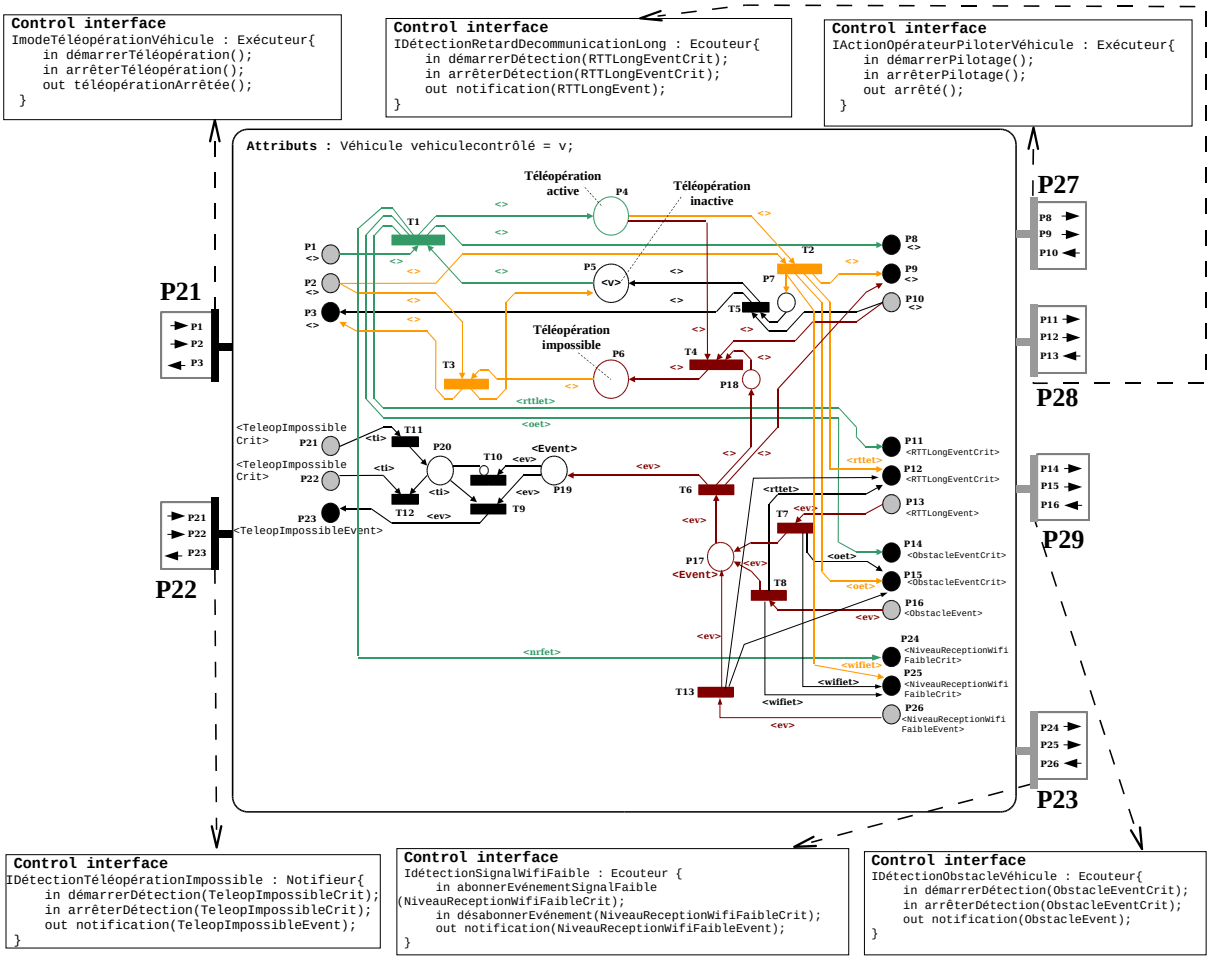


FIG. 9.15 – Le composant de contrôle `ModeVéhiculeTéléopéré`

d'un événement `TéléopérationImpossibleEvent`) ; un jeton banalisé est placé dans la place de sortie `P9` (demande d'arrêt de l'activité du composant de contrôle `ActionOpérateurPiloteVéhicule`).

Lorsqu'un jeton arrive dans la place `P21`, la détection d'événements signalant l'impossibilité de téléopérer le véhicule est activée. La transition `T11` est tirée et la place `P20` marquée, ce qui indique que le composant est en *mode de détection actif*. En cas d'arrêt du service de notification, la place `P22` est marquée et la transition `T12` est tirée, supprimant ainsi le jeton de la place `P20`. Lorsque cette place `P20` est marquée et qu'un jeton est déposé dans la place `P19`, un événement `TeleopérationImpossibleEvent` est généré par le tir de la transition `T9` et placé dans un jeton qui marque la place `P23`. L'événement `TeleopérationImpossibleEvent` est alors émis. Dans le cas contraire, `P20` est vide et la transition `T10` est tirée, supprimant ainsi l'événement (pas de notification). Lorsque le jeton contenant la confirmation d'arrêt arrive en place `P10`, la transition `T4` ou la transition `T5` est tirée. `T4` est tirée lorsque `P18` est marquée, c'est à dire lorsque un événement remettant en cause la téléopération a été notifié. Dans ce cas, `P4` est démarquée et `P6` marquée, ce qui correspond au passage du composant dans l'état *téléopération impossible*. Il ne sortira de cet état que lorsqu'une demande d'arrêt de `ModeVéhiculeTéléopéré` sera formulée : un jeton est placé dans `P2` et cela entraîne le tir de `T3`, qui va amener le démarquage de `P6`, le marquage de `P5` et la génération d'une confirmation d'arrêt. Dans le cas où le composant est dans un état actif, `P4` est marquée et c'est `T2` qui est tirée : ceci amène le dépôt d'un jeton dans la place de sortie `P9` (correspondant à l'arrêt de l'action), dans les places de sortie `P12` et `P15` (désabonnement à l'écoute d'événements) et dans la place `P7`. Dans le cas où `P7` est marquée, c'est la transition `T5` qui est tirée, ce qui va remettre le composant dans un état de *téléopération inactive* (marquage de la place `P5`) et amène le dépôt d'un jeton dans la place `P3` (émission de la confirmation de l'arrêt).

Le `ModeVéhiculeTéléopéré` est pour l'instant relativement simple, mais il pourrait être enrichi de réelles capacités de réaction à des événements rendant la téléopération impossible ou dangereuse. Par exemple, dans le cas de la détection d'un obstacle, il est envisageable de commuter d'une action de télélopération (`ActionOpérateurPiloteVéhicule`) vers une action prenant en charge le déplacement de façon autonome et assurant l'évitement d'un obstacle de façon automatique. Une fois l'obstacle évité, le pilotage serait redonné à l'opérateur via la commutation de cette action autonome vers l'action de téléopération.

9.3.11 Superviseur Mobile

Le composant de contrôle `SuperviseurMobile` (cf. fig. 9.16) gère la commutation entre le mode autonome et le mode téléopéré, en fonction des ordres de ressource qu'il reçoit. Il possède plusieurs ports fournis qui offrent l'accès à des services permettant de contrôler l'engagement des modes. `P11` et `P13` permettent d'engager respectivement les modes autonome et télé-opéré. Le port `P12` offre l'accès au service de désengagement du mode actif. Enfin le port `P14`, typé par l'interface `IDétectionDésengagementModeSpontanné`, offre l'accès au service de notification du désengagement spontanné du mode courant. `SuperviseurMobile` possède quatres ports requis qui lui permettent d'interagir avec les composants de contrôle *mode*. `P17` et `P18` lui permettent d'accèder aux services du composant `ModeVéhiculeTéléopéré`, alors que `P15` et `P16` lui permettent d'accéder aux service de `ModeVéhiculeAutonome`.

Le comportement asynchrone du `SuperviseurRessource` est décrit dans la figure 9.16. Trois places permettent de stipuler le mode actuellement en activité en fonction de leur marquage : lorsque `P9` est marquée, la ressource `Mobile` n'a aucun mode activé ; lorsque `P12` est marquée, la ressource `Mobile` est en mode autonome ; lorsque `P11` est marquée, la ressource `Mobile` est en mode téléopéré. La logique de l'engagement et du désengagement des modes est la même pour les modes téléopéré et autonome, c'est pourquoi nous ne décrirons que l'engagement et le désengagement du mode téléopéré.

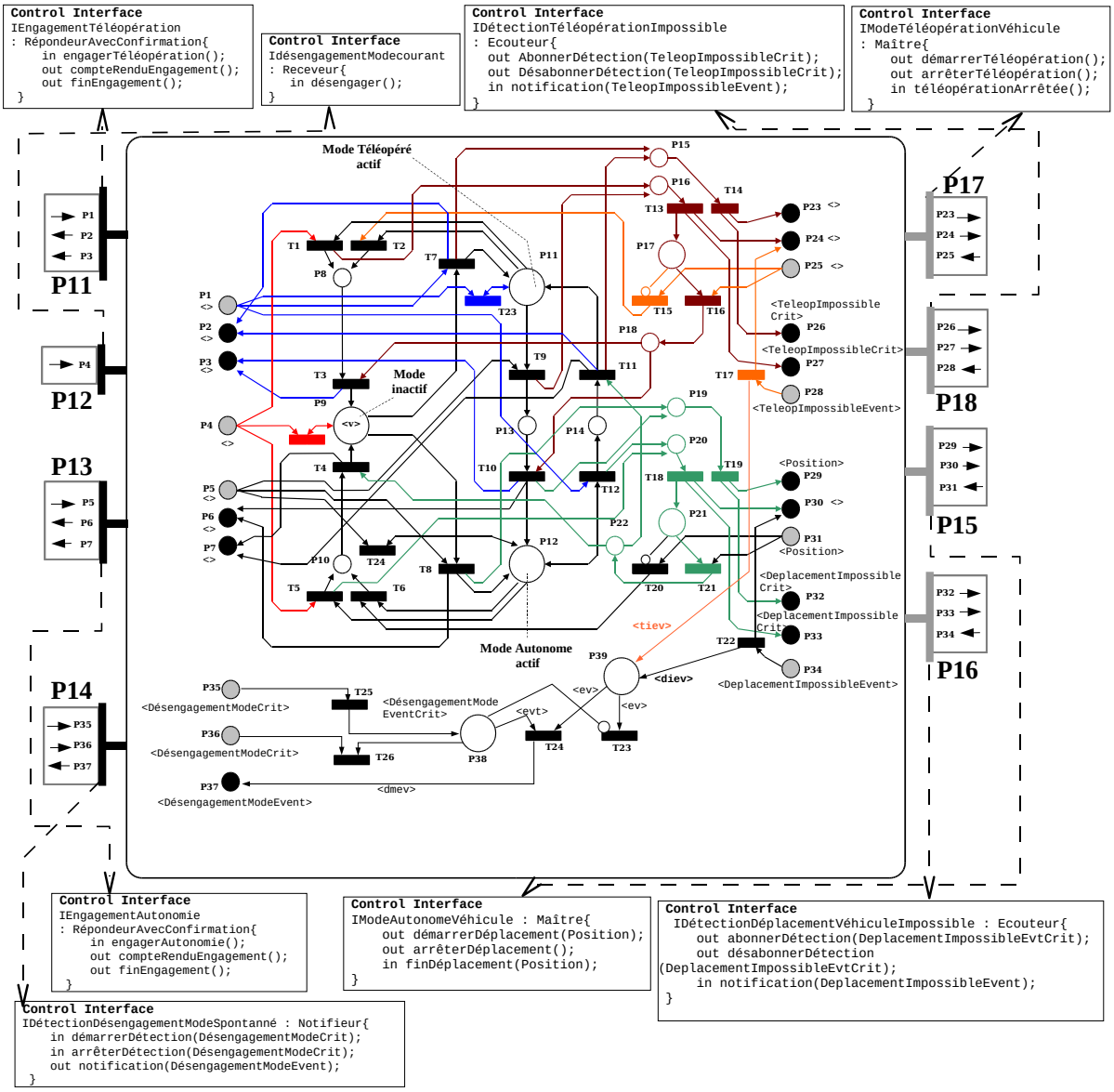


FIG. 9.16 – Le composant de contrôle `Superviseur Mobile`

A l'initialisation, aucun mode n'est actif (i.e. la place P9 est marquée). Lorsqu'un jeton arrive dans la place d'entrée P1, cela correspond à la réception d'un message lié au service IEngagementTéléopération. La transition T7 est alors tirée si aucun mode n'était actif (place P9 marquée), ce qui entraîne le dépôt d'un jeton dans la place P2, afin de renvoyer une confirmation d'engagement du mode téléopéré. Le tir de T7 amène également le dépôt d'un jeton dans la place P11, ce qui signifie que le mode de téléopération est maintenant actif. Le tir de T7 amène, enfin, le dépôt d'un jeton dans la place P15, ce qui entraîne le tir de T14 : un jeton est déposé dans la place P23, ce qui exprime une demande démarrage du mode téléopéré ; un jeton est déposé dans la place P26, ce qui exprime une demande d'abonnement à un événement TéléopérationImpossibleEvent. La transition T23 est tirée si le mode actif est déjà le mode de téléopération (place P11 marquée), ceci n'entraîne aucune action particulière. La transition T12 est tirée si le mode actif courant est le mode téléopéré (place P12 marquée). Cela entraîne alors le dépôt d'un jeton dans la place P20 et donc le tir de la transition T18. Ceci va entraîner le dépôt de jeton dans les places P30 et P33, ce qui correspond respectivement à une demande d'arrêt d'exécution du mode autonome et à un désabonnement à l'écoute d'événements DeplacementVéhiculeImpossibleEvent. Le tir de T12 amène aussi le dépôt d'un jeton dans la place P14. La transition T11 est tirée lorsque la confirmation d'arrêt du mode autonome est arrivée dans la place P22. Le tir de T11 amène le dépôt d'un jeton dans la place P11 (*mode de téléopération actif*) et dans la place P15 (démarrage du mode téléopéré, cf. plus haut). Le tir de T11 amène également le dépôt d'un jeton dans la place P7, afin de signaler la fin d'exécution du mode autonome.

Si un jeton arrive dans la place d'entrée P4, cela correspond à une demande d'arrêt du mode actif, quel qu'il soit. Dans le cas où le composant est en mode téléopéré, c'est la transition T1 qui est tirée, ce qui entraîne le dépôt de jetons dans les places P8 et P16. Le tir de la transition T13 amène le dépôt d'un jeton dans les places P24 et P27, pour signaler respectivement l'arrêt de l'activité du mode téléopéré et le désabonnement à l'écoute d'événements TéléopérationImpossibleEvent. Un jeton est placé dans la place P17, correspondant à l'attente de confirmation d'arrêt du mode téléopéré. Lorsque le jeton de confirmation arrive dans la place d'entrée P25, la transition T16 est tirée, et un jeton est déposé dans la place P18 pour stipuler que la désactivation du mode est effective. La transition T3 est alors tirée et le jeton de type Véhicule placé dans la place P9 (pas de mode actif). Un jeton est également déposé dans la place P3, afin d'émettre la confirmation d'arrêt de l'exécution du mode téléopéré.

Le désengagement des modes peut également être fait de manière automatique lorsque des événements notifient de l'impossibilité d'utiliser le mode. Lorsque le mode téléopéré est actif, l'écoute d'événements TeleopérationImpossibleEvent l'est également. Une jeton peut donc à tout moment arriver dans la place P28. Dans ce cas, la politique choisie est de désengager automatiquement le mode. Lorsque la place P28 est marquée la transition T17 est tirée : un jeton est alors déposé dans la place P24, ce qui correspond à une demande d'arrêt du mode téléopéré. Une confirmation d'arrêt est alors susceptible d'arriver dans la place P25 sans que pour autant la demande d'arrêt ne soit explicite (place P17 non marquée). Dans ce cas, T15 est tirée, ce qui amène le dépôt de jetons dans les places P35 et P18. Le mode téléopéré est désengagé via le tir des transitions T2 puis T3 : le composant est alors dans un mode "inactif".

Le désengagement d'un mode est une information importante pour le superviseur global du robot mobile. C'est pourquoi le composant de contrôle SuperviseurMobile propose un service de notification d'événements DésengagementModeEvent. Ces événements permettent de notifier lorsque le mode autonome ou le mode téléopéré vient d'être désengagé de façon automatique (sur arrivée d'événements DéplacementImpossibleEvent ou TéléopérationImpossibleEvent). La partie du RdPOC contenant les places P35 à P39 et les transitions T23 à T26 décrivent ce mécanisme de détection d'événements.

La gestion de la cohérence de l'engagement et du désengagement de modes est, dans cet exemple, relativement simple. Pourtant, il est possible d'imaginer des mécanismes beaucoup plus complexes. En

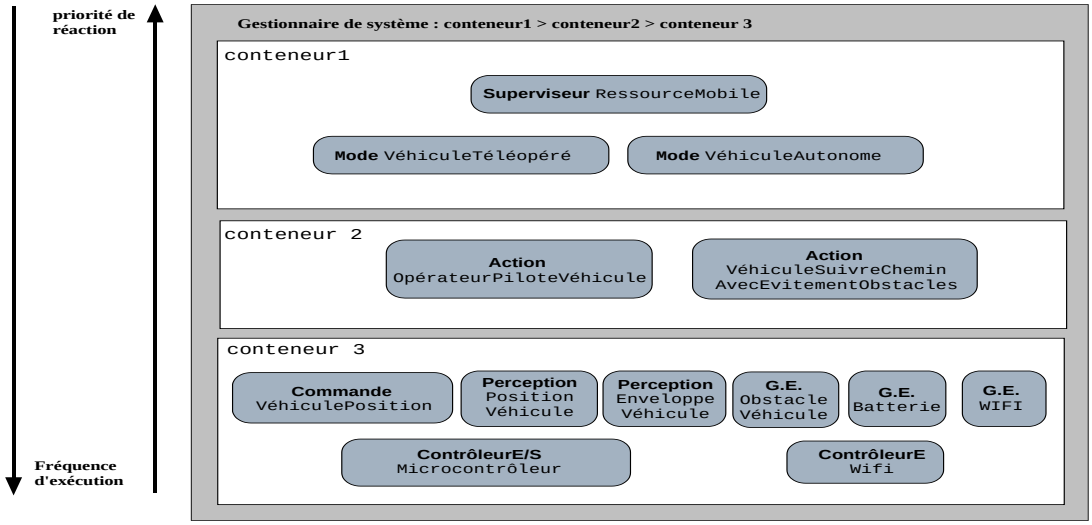


FIG. 9.17 – Exemple de déploiement de la ressource Mobile

particulier, il pourrait être intéressant de mettre en place un mécanisme de commutation automatique de mode : lorsque le `ModeVéhiculeAutonome` génère un événement `DéplacementVéhiculeImpossible`, la commutation se fait automatiquement vers le `ModeVéhiculeTéléopéré`, afin que l'opérateur prenne la main et sorte le véhicule de sa situation. Inversement, lorsque le `ModeVéhiculeTéléopéré` génère un événement `TéléopérationImpossibleEvent`, la commutation pourrait être faite automatiquement vers le `ModeVéhiculeAutonome` afin que le véhicule continue sa route (par exemple à partir de la dernière direction indiquée par l'opérateur) tout en évitant les obstacles.

9.3.12 Description du déploiement

La description du déploiement est la phase qui permet de prendre en compte le problème de la réactivité au sein du contrôleur, en paramétrant certaines propriétés de niveau système. Cette description du déploiement est influencée par l'organisation des composants de contrôle au sein de l'architecture et par les connecteurs utilisés. Dans notre cas, ce sont les composants et interactions utilisés pour la ressource `Mobile`. Cette description est aussi influencée par l'infrastructure matérielle sur laquelle elle est déployée. Puisque le robot mobile n'embarque qu'une carte PC embarquée, nous ne pouvons utiliser dans la description de l'infrastructure matérielle, qu'un seul gestionnaire de système. La description consiste alors à définir les conteneurs présents sur chaque gestionnaire de système et à leur affecter des composants de contrôle à exécuter. A partir de la notion de conteneur, nous possédons une unité de modélisation à partir de laquelle il est possible de gérer des priorités d'exécution entre groupes de composants (et plus largement leur ordonnancement). Nous pouvons par exemple choisir d'associer un conteneur à chaque couche de l'architecture ou un conteneur pour chaque composant de contrôle. Nous n'avons pas défini de règle absolue pour le déploiement, néanmoins nous proposons une démarche simple basée sur le modèle d'organisation d'une architecture.

– Le premier principe est de regrouper tous les composants de contrôle conceptuellement situés dans une même couche, dans un même *conteneur*, lorsqu'ils sont déployés sur un même *gestionnaire de système*. Par exemple toutes les actions sont contenues dans le `conteneur 2`. Ceci se justifie par le fait que l'`ActionOpérateurPiloteVehicule` doit réagir plus prioritairement

que le composant `CommandeVéhiculePosition`, qui se trouve dans le `conteneur 3`.

- Le deuxième principe est que les composants de contrôle s'exécutant de façon plus fréquente, se trouvent dans les conteneurs représentant les couches basses de l'architecture. Par exemple, le fonctionnement des composants *commande* (périodique) est plus fréquent que celui des composants *action*, ils sont donc placés dans le conteneur de plus faible priorité (`conteneur 3`). Par exemple, le conteneur contenant les composants *commande* sera moins prioritaire que le conteneur contenant les composants *action*.
- Le troisième principe est que les conteneurs sont ordonnés en fonction de la couche qu'ils représentent. L'ordre des conteneurs est décrit de façon décroissante, des conteneurs les plus prioritaires pour les couches hautes, vers les conteneurs les moins prioritaires pour les couches basses.
- Enfin, le dernier principe consiste à déployer les composants de contrôle relatifs à une ressource sur un même gestionnaire de système si cela est réalisable (i.e. si les capacités d'exécution du noeud matériel représenté par le gestionnaire de système le permettent), afin de limiter les communications réseaux entre ces composants et ainsi éviter une perte de temps significative dans la transmission des messages. Ceci est particulièrement important dans les bas niveau de l'architecture (actions, commandes, générateurs d'événements), là où la fréquence des interactions entre composants est potentiellement grande et où la nécessité de réagir vite est primordiale. Dans l'exemple, il n'y a pas d'infrastructure matérielle distribuée, ce qui règle le problème, le gestionnaire de système est donc utilisé pour déployer l'intégralité de la ressource `Mobile`.

La figure 9.17 présente un exemple de déploiement de l'architecture logicielle. Dans la solution de déploiement retenue, les composants de contrôle sont regroupés dans trois conteneurs : le `conteneur 1` contient les composants *mode* et le `SuperviseurMobile` ; le `conteneur 2` contient les composants *action* ; le `conteneur 3` contient les composants *commandes*, *générateurs d'événements*, *perceptions* et *contrôleurs d'entrées/sorties*. Nous supposons que les mécanismes de planification du `ModeVéhiculeAutonome` ont été définis comme s'exécutant en "temps-restant global" (cf. chapitre 9), afin que leur exécution ne vienne pas bloquer celle des conteneurs de plus bas niveau.

9.4 Ressource Manipulateur

Dans l'optique d'ajouter un bras mécanique au robot mobile qui a fait plus particulièrement l'objet de notre étude, nous présentons dans cette section la modélisation proposée pour la ressource `Manipulateur`. Nous illustrons, qui plus est, les capacités d'évolution d'un contrôleur, par ajout de nouvelles ressources, à mesure que la partie matérielle du robot est enrichie de nouveaux éléments. Nous généralisons la modélisation du `Manipulateur` pour un bras mécanique à six degrés de liberté, que nous pourrons, au besoin, simplifier en fonction du bras mécanique qui sera réellement embarqué sur le robot.

9.4.1 Architecture

La ressource `Manipulateur` est une configuration, présentée dans la figure 9.18. Elle est composée d'un seul superviseur de ressource (`superviseurManipulateur`) et d'un seul mode, puisqu'elle ne fonctionne ici qu'en mode autonome. Le `ModeBrasAutonome` est capable de réagir à un ensemble d'ordres (de ressource). Il est capable de décomposer chaque ordre en une succession d'actions à réaliser. Par exemple, pour déplacer un objet dans son espace atteignable, il doit tout d'abord contrôler le bras mécanique afin qu'il se rapproche de la position de l'objet (`ActionDéplacerBras`), le faire entrer en contact avec l'objet (`ActionBrasRechercherContact`), lui faire saisir l'objet (`ActionPréhenseurSaisir`), le déplacer à nouveau, puis lui faire relâcher l'objet à la position cible (`ActionPréhenseurRelâcher`). Pour réaliser chaque ordre, il fait donc une succession d'appels à des composants de contrôle

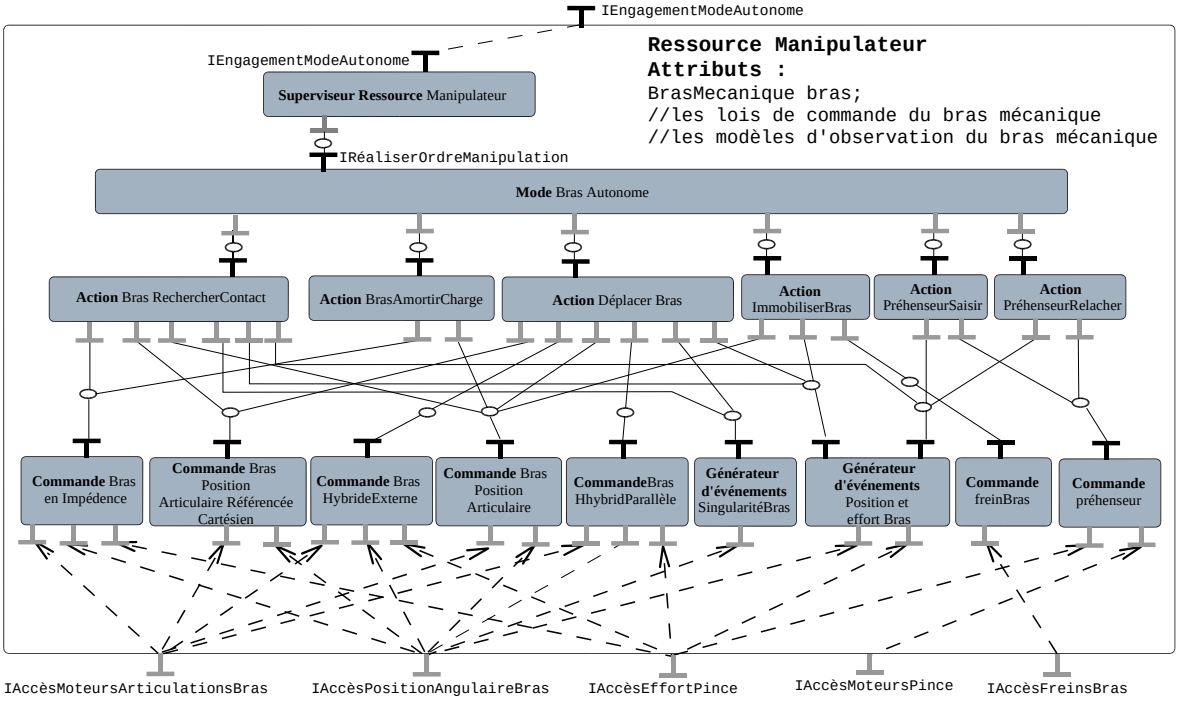


FIG. 9.18 – Architecture logicielle d'une ressource Manipulateur

action en charge de la réalisation de ces différentes actions. Les différents composants de contrôle *action*, utilisés au sein de la ressource `Manipulateur`, sont les suivants :

- `ActionBrasRechercherContact` permet au bras mécanique de rechercher un contact avec un objet ou une paroi qui se trouve dans son espace atteignable.
- `ActionBrasAmortirCharge` permet au bras mécanique de compenser la charge d'un objet lourd qu'il porte (pour un transport amorti).
- `ActionDéplacerBras` permet au bras mécanique de suivre un chemin dans son espace atteignable.
- `ActionPréhenseurSaisir` permet au préhenseur du bras mécanique de saisir un objet qui se trouve dans son espace atteignable.
- `ActionPréhenseurRelâcher` permet au préhenseur du bras mécanique de relâcher un objet à une position donnée dans son espace atteignable.
- `ActionImmobiliserBras` permet de stabiliser le bras mécanique dans une position, puis de mettre ses freins avant d'arrêter son asservissement. La demande de terminaison de cette action entraîne la reprise de son contrôle en position stable puis dans l'enlèvement des freins du bras mécanique. C'est l'action supposée active à l'initialisation et à la terminaison de la mission. Elle peut être utilisée

9.4.2 Commandes

Chacun de ces composants de contrôle *action* se base sur un ensemble de composants de contrôle *commande* disponibles pour asservir le `bras mécanique`. Un composant *commande* agrège et exploite un composant de représentation *loi*, représentant la loi de commande utilisée. Chaque composant de représentation *loi* possède un port fourni permettant d'accéder à sa fonctionnalité de calcul de la

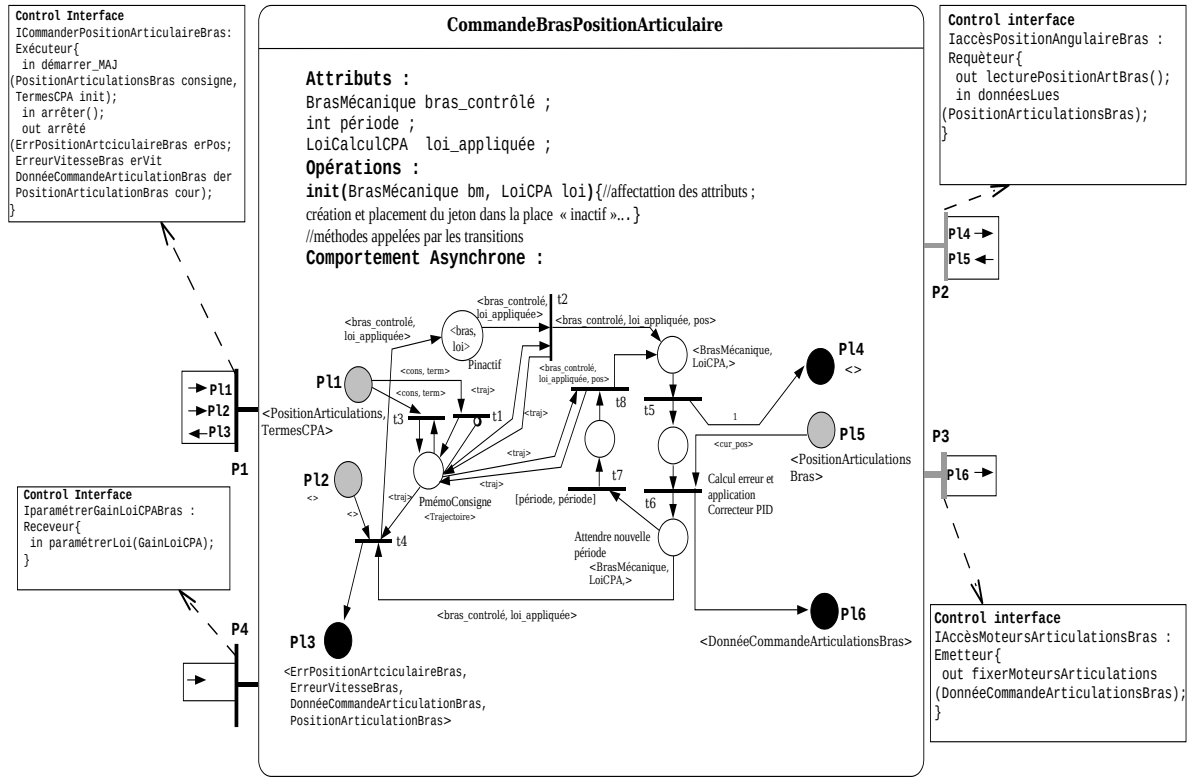


FIG. 9.19 – Exemple simplifié du composant de contrôle `Commande Bras Position Articulaire`

donnée de commande à générer (en fonction des données capteurs et d'une consigne). Il possède un port requis qui lui permet d'accéder aux fonctionnalités et à l'état de l'élément matériel auquel cette *loi* s'applique. Par exemple il existe un ensemble de composants de représentation *loi* qui ont chacun un port requis connecté avec un port fourni du composant de représentation `bras mécanique` : `loiCalculcommandePositionArticulaire` (`loiCalculCPA`), `loiCalculCommandeImpédance` (`loiCalculCIMP`), `loiCalculCommandeHybrideParallèle` (`loiCalculCHP`) [34], etc. Un composant *commande* est chargé de contrôler la génération de données de commande par une *loi* (par exemple en paramétrant la *loi* ou en fixant la périodicité de l'appel à sa fonctionnalité de génération de données de commande).

La figure 9.19 présente un composant de contrôle `CommandeBrasPositionArticulaire` qui met en œuvre l'asservissement permettant de positionner le `bras mécanique` via des consignes de positions dans son espace articulaire. Ce composant a pour attributs un composant de représentation `loiCalculCPA` et un composant de représentation `Bras Mécanique` connectés entre eux. La `loiCalculCPA` utilisée étant relativement peu coûteuse en terme de temps de calcul, elle est utilisée pour freiner et stabiliser le `bras Mécanique` (compensation de la gravité). Le composant de contrôle `CommandeBrasPositionArticulaire` a pour attribut une `période` qui permet de paramétrer la cadence à laquelle il applique la `loiCalculCPA`. Il possède un port fourni `P1` qui permet à d'autres composants de contrôle d'accéder à son service d'asservissement (`ICommanderPositionArticulaireBras`). Son port fourni `P4` donne la possibilité de paramétrer les gains de la loi de commande appliquée. Il interagit à travers ses deux ports requis `P2` et `P3`, avec un composant de contrôle *contrôleur d'entrées/sorties* qui fournit l'interface `IAccèsPositionsArticulationsBras` et avec un *contrôleur d'entrées/sorties* qui fournit l'interface `IAccèsMoteursArticulationsBras` (dans notre cas il s'agit du

ContrôleurE/SMicrocontrôleur).

Le RdPOC décrivant le comportement asynchrone de CommandeBrasPositionArticulaire définit plusieurs états exclusifs. Ces états sont représentés par différentes places. La place Pinactif représente l'état dans lequel le composant n'a pas d'activité de contrôle. Lorsque le composant est inactif, cette place est marquée avec un jeton contenant les deux attributs de CommandeBrasPositionArticulaire, à savoir bras_contrôlé et loi_appliquée (correspondant au jeton <bras, loi>). Dans cet état, il est en attente de demande de démarrage de génération de commande, provenant de la place Pl1. Lorsqu'il reçoit une demande de démarrage, une consigne exprimant la position articulaire finale désirée est passé en paramètre, ainsi que le jeu de termes (erreurs de position et de vitesse, et terme intégral) qui va servir à son initialisation (pour la compensation de la gravité). La transition t1 est tirée, ce qui déclenche l'appel d'une méthode de génération de points de consigne pour donner une trajectoire au bras mécanique [36], puis le dépôt d'un jeton Trajectoire dans la place PMémoConsigne ce qui entraîne le tir de la transition t2. La place initialement marquée est alors vidée et le composant passe dans un état "actif". Lorsqu'il est "actif", il applique périodiquement la loiCalculCPA sur le bras mécanique afin de générer des données de commande. Cette boucle de commande est décomposée en plusieurs étapes qui correspondent aux transitions t5, t6, t7 et t8 (figure 9.19) et aux états intermédiaires représentés par les différentes places à droite sur la figure. La première étape (transition t5) consiste à récupérer les données de position courante du bras mécanique, via le port requis typé par l'interface IAccèsPositionsAngulairesBras (envoi d'un message après dépôt d'un jeton dans la place Pl4). Une fois ces données reçues (arrivée d'un jeton dans la place Pl5), la deuxième étape (transition t6) consiste à calculer l'écart entre la position courante et la position intermédiaire désirée pos, afin de connaitre l'erreur de vitesse et l'erreur de position. Une fonction correcteur PID (contenue dans le composant loiCalculCPA) est alors appelée afin de calculer la DonnéeCommandeArticulationsBras générée. Cette DonnéeCommandeArticulationsBras est ensuite déposée dans la place de sortie Pl5 ce qui correspond à l'émission d'un message fixerMoteurArticulations). Cette DonnéeCommandeArticulationsBras est donc envoyée au ContrôleurE/SMicrocontrôleur qui l'applique au moteurs du bras mécanique. La troisième étape (tir de t7) consiste à attendre un temps donné, calculé à partir de l'attribut période, avant de relancer la génération d'une donnée de commande (on peut déduire ce temps d'attente en soustrayant à période le temps de calcul du cycle précédent). La dernière étape (transition t8) consiste à relancer l'exécution en prenant un nouveau point de consigne intermédiaire désiré, contenu dans le composant de représentation Trajectoire. Précisons que les relations avec les services d'actualisation (interaction exprimées par les interfaces des ports requis sur la figure) des données capteurs (respectivement actionneurs) sont non bloquantes, au sens où la dernière valeur connue (respectivement calculée) est retournée (respectivement émise).

Sur une demande d'arrêt (arrivée d'un jeton dans la place d'entrée Pl2), le composant est remis dans un état "inactif" (tir de t4) et renvoi un compte rendu de son exécution (dépôt d'un jeton dans la place de sortie Pl3). Ce compte rendu est un jeton contenant les composants de représentation représentant : l'erreur de position courante, l'erreur de vitesse courante, la dernière donnée de commande appliquée aux actionneurs et la position courante du bras mécanique. Ce compte rendu peut-être, par exemple, utilisé par des composants de contrôle *action* afin de calculer l'initalisation d'un autre composant *commande*, si une commutation de loi de commande doit avoir lieu.

9.4.3 Action Bras Rechercher Contact

Le composant de contrôle ActionBrasRechercherContact, présenté dans la figure 9.20, est un exemple de composant *action*, qui utilise le composant de contrôle CommandeBrasPositionArticulaire. Il met en œuvre l'activité permettant au bras mécanique de rechercher le contact avec une surface quelconque (la direction du contact étant indiquée via un paramètre), supposée être dans son espace atteignable. Son port fourni typé par l'interface IActionBrasRechercherContact permet de contrô-

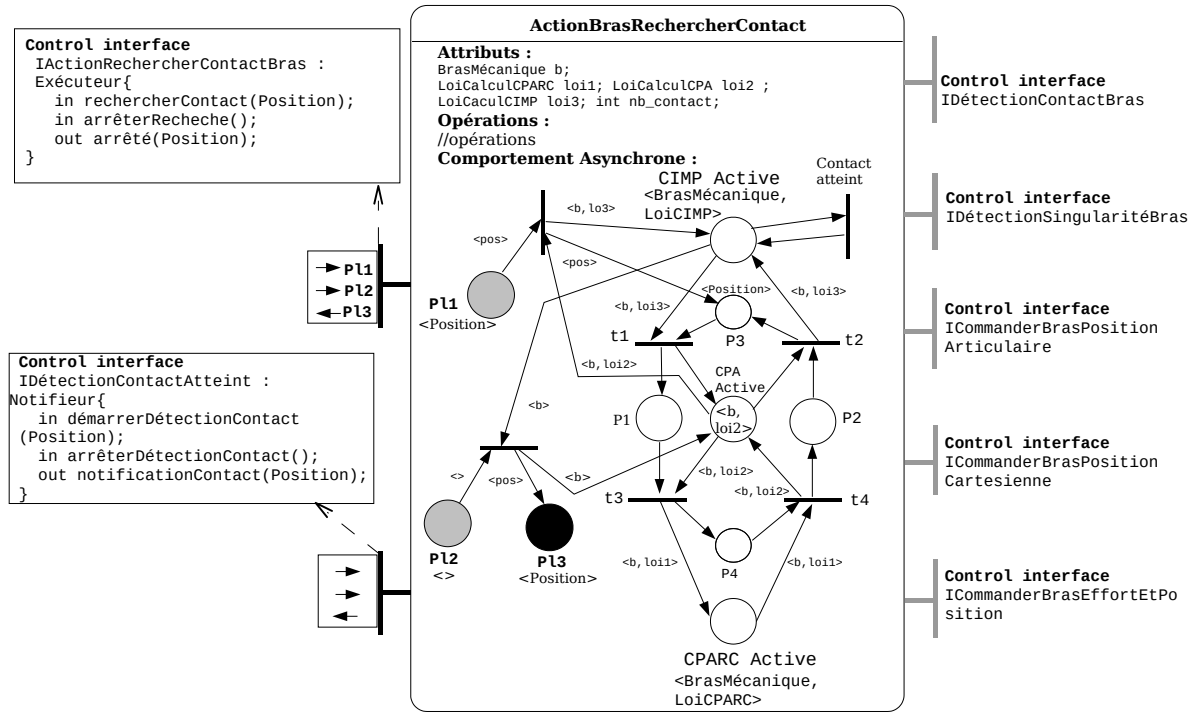


FIG. 9.20 – Exemple simplifié du composant `Action Bras Rechercher Contact`(a)

ler l'exécution de l'action. Son port fourni typé par l'interface `IDétectionCOntactAtteint` permet de notifier sa réalisation. Ses ports requis sont typés par l'ensemble des interfaces contenant les services dont il a besoin pour réaliser ses fonctionnalités : détection de contact (`IDétectionContactBras`), détection d'approches ou de sorties de configurations singulières (`IDétectionSingularitéBras`), commande en position articulaire (`ICommanderBrasPositionArticulaire`) ou en position cartésienne (`ICommanderBrasPositionCartesienne`), commande en effort et position (`ICommanderBrasEffort EtPosition`).

Ce composant *action* met en œuvre un mécanisme de commutation entre trois composants *commande* présents dans la ressource `Manipulateur` :

- `CommandeBrasPositionArticulaire (CPA)` est utilisé afin d'asservir en continu le bras mécanique (compensation de la gravité), afin qu'il se stabilise. Il sert à l'initialisation et à la terminaison de toutes les *actions* afin d'assurer la stabilité du bras lors du changement d'action. Il sert également à réaliser la transition entre deux composants *commande* au moment d'une commutation. Dans l'exemple d'`ActionBrasRechercherContact`, `CPA` est utilisé pour les commutations entre les deux autres composants *commande*, présentés ci-après.
- `CommandeBrasImpédance (CIMP)` permet d'asservir le bras mécanique avec un comportement de type amortisseur. Il est aussi utilisé pour mettre en œuvre la recherche de contact du bras mécanique avec une paroi ou un objet. La limitation de cette *commande* est qu'elle est sensible aux singularités. Ainsi les *actions* interagissant avec elle doivent prendre en compte la gestion des configurations singulières. Ce composant possède un port fourni référençant l'interface `ICommanderBrasEffortEtPosition`.
- `CommandeBrasPositionArticulaireRéférencéeCartésien (CPARC)` permet d'asservir le bras mécanique avec un contrôle en position. Cette *commande* est basée sur une *loi* insensible aux singularités (basée sur une consigne en espace cartésien, avec un changement de repère assurant

l'exclusion des points singuliers), c'est pourquoi elle est utilisée par `ActionBrasRechercherContact` pour sortir le bras mécanique hors d'une configuration singulière, avant de pouvoir poursuivre la recherche de contact avec `CIMP`. Ce composant possède un port fourni typé par l'interface `ICommanderBrasPositionCartésienne`.

`CPA` est le composant de contrôle *commande* utilisé pour faire la transition entre l'activité du composant de contrôle `CIMP` et celle du composant de contrôle `CPARC`. `ActionBrasRechercherContact` utilise `CPA` pour freiner (puis stabiliser) au plus vite le bras mécanique (commutation "à chaud"). Une fois le bras mécanique stabilisé, une commutation vers l'une ou l'autre des *commandes* `CIMP` ou `CPARC` est réalisée (commutation "à froid"), suivant les conditions (respectivement configuration singulière franchie ou à franchir). Les différentes commutations exprimées sur la figure 9.20 sont les suivantes :

- La transition `t1` exprime une commutation de `CIMP` vers `CPA`. Elle intervient lorsque la *commande* `CIMP` est active et lorsque l'approche d'une configuration singulière est détectée. Cette commutation directe entre ces deux composants *commande* a pour but d'arrêter et de stabiliser le bras mécanique contrôlé. Un jeton est alors déposé dans la place `P1`.
- La transition `t3` exprime une commutation de `CPA` vers `CPARC`. Elle intervient lorsque le bras mécanique est stabilisé, et permet d'activer `CPARC` qui va permettre de contourner les singularités. Le calcul de la trajectoire permettant de réaliser ce contournement n'est plus temporellement contrainte, puisque le bras mécanique est stabilisé.
- La transition `t4` exprime une commutation de `CPARC` vers `CPA`. Elle intervient lorsque le bras mécanique a été positionné hors d'une configuration singulière et peut donc reprendre son activité initiale de recherche de contact. Cette commutation a lieu afin de stabiliser le bras mécanique avant de reprendre la recherche de contact via `CIMP`.
- La transition `t2` exprime une commutation de `CPA` vers `CIMP`. Elle intervient une fois le bras mécanique stabilisé afin de reprendre la recherche de contact.

Pour des raisons de clarté du dessin, les différentes étapes d'une commutation sont agrégées dans une même transition. Pourtant, chacune de ces commutations nécessite des synchronisations avec les composants de contrôle *commande* commutés et les composants de contrôle *générateurs d'événements* qui notifient les événements pertinents (par exemple `SingularitéEvent` et `PositionAtteinteEvt`). Les transitions `t1` à `t4` sont donc en fait constituées de plusieurs étapes, traduisant les synchronisations entre le composant `ActionBrasRechercherContact` et les composants de contrôle avec lesquels il est en interaction. La figure 9.21 montre la structure réelle de la transition `t1`. La commutation supportée par `t1` est déclenchée par l'arrivée, dans la place d'entrée `IDétectionSingularitéBras.notification`, d'un jeton contenant un composant de représentation `SingularitéEvent`. L'`ActionBrasRechercherContact` demande alors l'arrêt du composant `CommandeBrasImpédance`. Puis elle se met en attente de la réception de la confirmation d'arrêt du service `ICommanderBrasEffortEtPosition` (assurée par le composant de contrôle `CommandeBrasImpédance` dans notre architecture). Lorsque cette confirmation arrive dans la place `ICommanderBrasEffortEtPosition.arrêté`, la seconde transition est tirée, ce qui entraîne l'appel de l'opération `calculTILoiCPA`. Cette opération initialise le jeu de termes nécessaire à l'exécution de `CommandeBrasPositionArticulaire`, à partir des erreurs de position et de vitesse et de la dernière donnée de commande appliquée aux actionneurs du bras mécanique (renvoyés par la confirmation d'arrêt de `CommandeBrasImpédance`). Ces termes vont permettre d'initialiser `CommandeBrasPositionArticulaire` afin qu'elle compense la gravité en fonction de la dynamique du bras, lors de la comutation.

La dernière étape de cette commutation consiste alors à démarrer la *Commande* `CPA` qui est chargée de stabiliser le `Bras mécanique`. Ceci se traduit par le dépôt d'un jeton contenant la consigne finale désirée de type `PositionsArticulationsBras` (c'est-à-dire la position actuelle du bras puisque `CommandeBrasPositionArticulaire` est utilisé pour freiner le bras), ainsi que du jeu de termes `TermesCPA` calculé, dans la place de sortie `ICommanderPositionArticulaireBras.démarrer_MAJ`). Les places `P1`, `P2`, `P3` et `P4` (cf. fig. 9.20) permettent de stocker l'objectif à long terme pour le

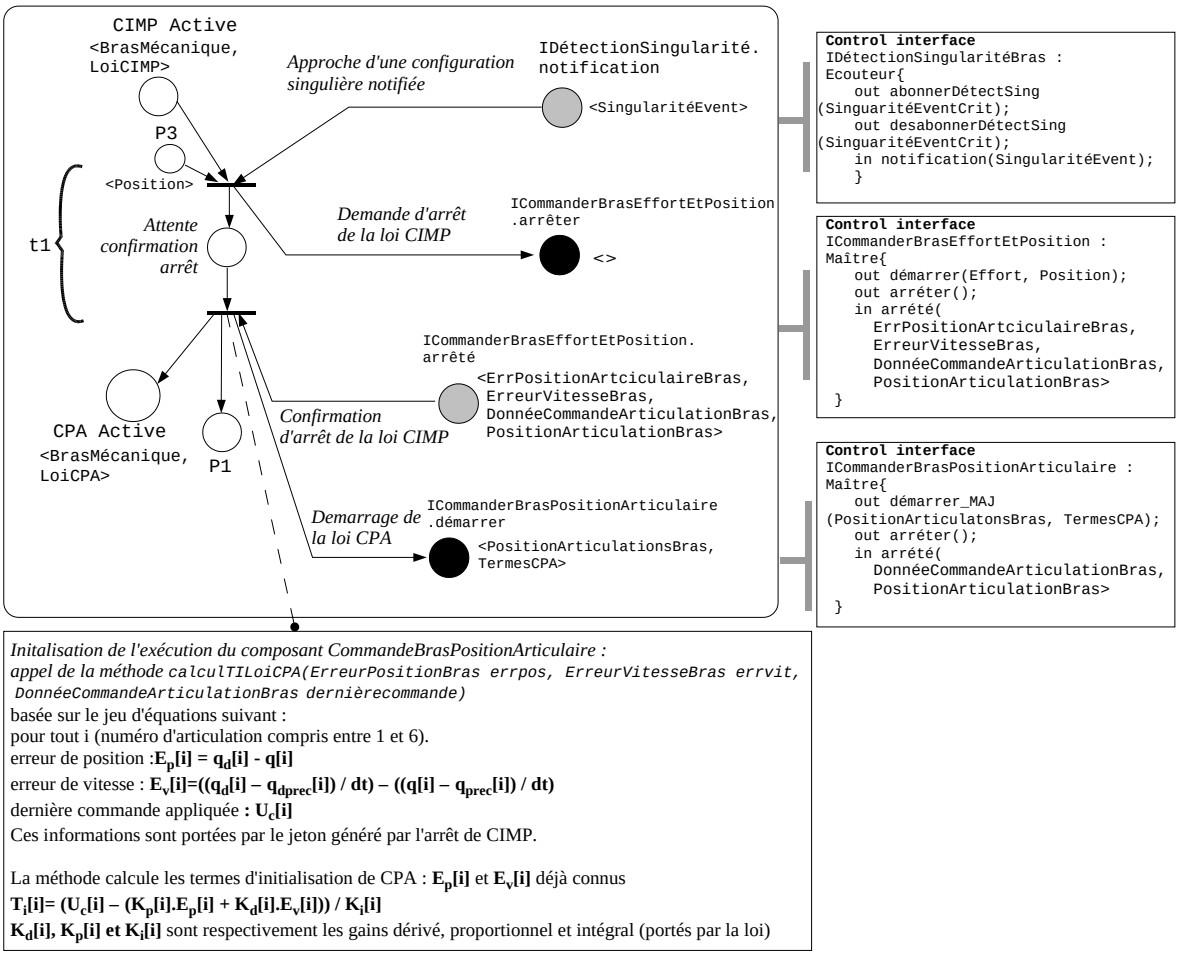


FIG. 9.21 – Exemple simplifié d'un composant `ActionBrasRechercherContact`(b) : Commutation de CIMP vers CPA

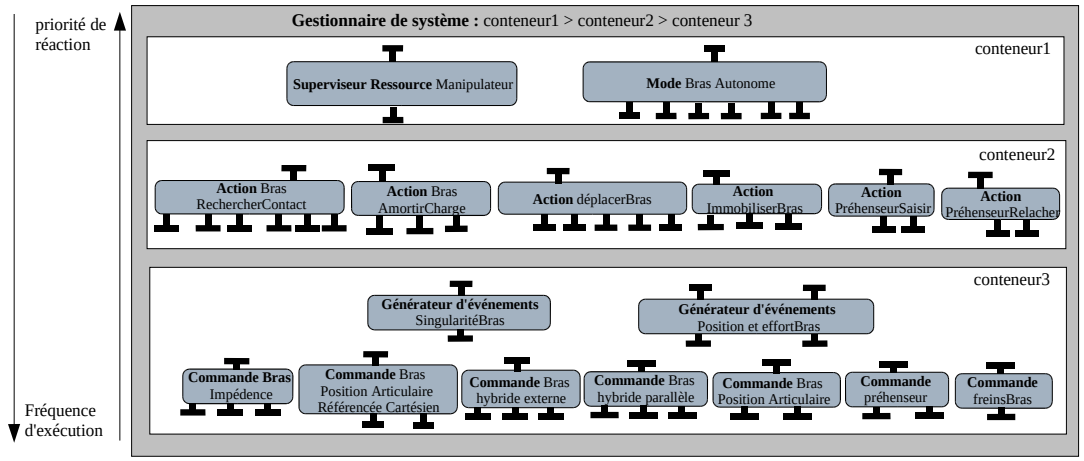


FIG. 9.22 – Exemple de déploiement de l'architecture de la ressource Manipulateur

composant *action*, à savoir la consigne de position finale (composant de représentation de type `Position`), elle-même fixée lors de l'arrivée d'un jeton dans la place `Pl1` (réception du message `IActionRechercheContactBras.rechercherContact`). Cette consigne est utile au moment des commutations, afin que l'action puisse prendre certaines décisions liées à l'accomplissement de l'objectif que doit réaliser le composant *action*. Par exemple, la consigne de position à atteindre est stockée dans `P3` (cf. fig. 9.21) au moment d'une commutation entre `CPA` et `CIMP` afin que le composant mémorise sont objectif final à atteindre, dans le cas où une commutation aurait lieu. Lorsqu'une singularité survient, ce "jeton mémoire" est propagé vers `P1`. Il est utilisé, au niveau de `t3`, pour trouver un point à atteindre hors de la zone singulière qui rapproche le bras mécanique de l'objectif final.

Le composant de contrôle `ModeBrasAutonome` utilise l'ensemble des actions disponibles pour contrôler le bras mécanique, afin de réaliser des ordres provenant des couches plus hautes de l'architecture. Pour cela il possède un mécanisme de décision interne qui, en fonction d'un ordre donné, va décomposer cet ordre en une série d'actions à réaliser. Dans notre approche actuelle, le séquencement des actions à réaliser est prédéfini en fonction des ordres à réaliser. Il peut être remis en cause et adapté en fonction d'événements reçus par le mode, mais la séquence de rattrapage correspondante est elle-même prédéfinie (sur la structure du modèle). Une approche plus poussée pourrait intégrer l'utilisation d'un planificateur d'actions pour générer ce séquencement ainsi que les séquences de rattrapage correspondantes.

9.4.4 Description du déploiement

Le déploiement de la ressource `Manipulateur` sur un gestionnaire de système, est montré dans la figure 9.22. Les composants de contrôle y sont déployés dans trois conteneurs différents en fonction de leur nature : les *commandes* et les *générateurs d'événements* sont regroupés dans le conteneur le moins prioritaire (`conteneur 3`) ; les *actions* sont regroupées dans le conteneur de priorité intermédiaire (`conteneur 2`) ; le *mode* et le *superviseur ressource* sont regroupés dans le conteneur le plus prioritaire (`conteneur 1`). Les conteneurs 2 et 3 représentent les deux couches chargées de gérer les asservissements et les réactions critiques (e.g. évitement d'obstacles, franchissement de configurations singulières). Le conteneur 1 contient les composants encapsulant les mécanismes de délibération les plus complexes et les moins contraints en temps (planification d'actions). Nous supposons que les

mécanismes de planification de ces composants sont considérés comme s'exécutant en "temps restant global" (cf. chapitre 9). Ceci permet d'éviter que ces mécanismes, dont le temps de calcul est potentiellement long comparé à la dynamique des autres conteneurs, ne bloquent l'exécution des conteneurs gérant les réactions "réflexes". Cet exemple est uniquement illustratif, et il pourrait se révéler utile par exemple de déporter le conteneur 1 sur un autre gestionnaire de système prévu à cet effet, si la charge CPU le nécessite.

9.5 Conclusion

Nous avons présenté l'architecture logicielle du contrôleur de robot manipulateur mobile en cours de développement. A travers l'exemple de la ressource `Manipulateur`, nous avons illustré le fonctionnement des couches basses de l'architecture, où se réalisent les commutations de commandes. A travers l'exemple de la ressource `Mobile`, nous avons plus particulièrement illustré un exemple de conception d'un mode téléopéré. Notre objectif à court terme est de passer à la phase de programmation de cet exemple, afin de valider expérimentalement cette modélisation. Il faut pour cela disposer d'un environnement logiciel performant qui permette d'analyser les RdPOC, voire même d'effectuer des tests par la simulation, afin de détecter au plus tôt les possibles erreurs de conception.

L'utilisation des RdPOC a permis de décrire de façon précise et lisible les comportement asynchrones des composants de contrôle et connecteurs. L'approche par composants a permis de significativement réduire la complexité des modèles RdPO vis-à-vis des travaux de modélisation du contrôle par RdPO déjà menés au LIRMM. Elle a permis une description locale des comportements des composants, sans prendre en compte l'intégralité du modèle RdPO d'un contrôleur. Il est possible de penser qu'un découpage à granularité plus fine que celui proposé dans notre modèle d'organisation des contrôleurs, permettrait de décrire des entités dont les RdPOC seraient moins complexes. Néanmoins, un tel découpage pourrait également entraîner de plus nombreux et complexes assemblages de composants de contrôle. Il s'agit donc de trouver un "juste milieu" dans le découpage des architectures. En cela, une des problématiques futures sera de répondre à un découpage à grain plus fin de la couche "haute" d'une architecture (superviseur global et gestionnaire de mission). Différents types de composants gérant la ou les mémoires globales "à long terme" du robot devraient en particulier être définis (e.g. mémoire de l'environnement dans une carte, mémoire du déroulement de la mission). La plate-forme opérateur, qui est le composant via lequel l'opérateur se place comme l'organe de décision de plus haut niveau dans la mission, devrait également être détaillée.

Cette modélisation nous a aussi fait prendre conscience d'une problématique majeure : la méthodologie de définition des interfaces typant les ports des composants. La définition des interfaces a été réalisée, dans cet exemple, en fonction de l'architecture que nous voulions modéliser. Or, le paradigme des composants incite à la définition d'interfaces de façon indépendante vis-à-vis d'une application particulière, dans le but de standardiser les interactions entre composants. La standardisation est envisageable pour les composants de représentation et leurs interfaces, car les descripteurs de composants de représentation sont indépendants de l'utilisation qui en est faite au sein d'un contrôleur. De même, il semble tout à fait envisageable de définir des interfaces de rôle et des connecteurs standards, car les descripteurs de connecteurs peuvent être découplés d'un domaine applicatif particulier. A contrario, définir des interfaces de contrôle, a priori, sans prendre en compte l'application visée et son architecture, est beaucoup plus difficile. Nous n'avons pas de réponse globale à la problématique de la définition des interfaces et rien ne semble émerger des recherches en cours. Pourtant, au vu de notre expérience, le succès d'une approche à composants repose en grande partie sur la capacité des développeurs à utiliser et définir des interfaces standards.

Chapitre 10

Conclusion et Perspectives

Nous avons proposé au travers de ce manuscrit, une méthodologie de développement de contrôleurs de robots basée sur les composants logiciels. Ce travail s'est révélé être une véritable aventure entre deux "mondes" qui possèdent leur propre jargon et leur propre façon de voir les choses : celui des architectures de contrôle en robotique et celui des composants en génie logiciel. Un des grands enjeux de cette thèse était de faire cohabiter ces deux thématiques, ce qui s'est révélé d'autant plus complexe qu'elles sont encore loin d'être "stabilisées" autour de définitions et de propositions abouties et globalement acceptées. Le travail effectué pendant la thèse constitue, avant toute chose, une avancée notable pour la mise en cohésion de ces deux thématiques.

La méthodologie CoSARC repose sur un principe simple : séparer le modèle d'architecture de contrôle de la plate-forme choisie pour la mise en œuvre des architectures logicielles. Un modèle d'architecture de contrôle précise les différents types d'entités logicielles présentes dans une architecture et définit leurs responsabilités dans le système de prise de décision et leurs interaction potentielles. Une plate-forme de développement de contrôleurs logiciels de robots est adaptée aux problématiques du domaine : gestion de propriétés temps-réel de niveau système, qualité de conception, description des flots de contrôle avec des notations adaptées, etc. Elle possède des propriétés favorisant la réutilisation et l'intégration maîtrisée de briques logicielles, ce qui est essentiel pour un véritable essort de l'ingénierie logicielle en robotique. Ce principe de séparer les aspects conceptuels des aspects techniques, est une innovation en soi pour le monde des architectures de contrôle en robotique, dans lequel de nombreux travaux ne dissocient pas les deux aspects.

La méthodologie CoSARC concrétise ce principe en s'appuyant sur un modèle d'organisation d'architecture débarassé de considérations techniques, au bénéfice de concepts génériques. Ces derniers définissent un vocabulaire neutre vis-à-vis d'un domaine d'application robotique, à partir duquel "discuter" d'un contrôleur de robot et autour duquel est menée la phase d'analyse. Le modèle d'architecture a été pensé pour améliorer la lisibilité, en associant ces concepts à des types d'entités logicielles constituant les architectures de contrôle. Ces entités représentent les connaissances que le contrôleur possède sur le "monde" du robot : parties mécaniques, loi de commande, modèles d'observation, environnement, événements, etc. Elles représentent également différentes activités du robot, comme la gestion des modes de fonctionnement, la commutation d'action, l'observation d'événements et l'application de commandes. Le modèle présente deux visions complémentaires de l'organisation d'une architecture de contrôle : la vision hiérarchique, à travers le concept de couche, et la vision systémique, à travers le concept de ressource robotique. Il est basé sur une approche "mixte" qui couple la hiérarchisation de la prise de décision avec une réactivité maximale, via le "saut" de couche lors du transfert de l'information. D'autre part, il ouvre la porte à une intégration plus poussée du modèle d'organisation des architectures comportementales. Ce modèle est une contribution notable à l'établissement d'un consensus sur la façon d'organiser une architecture de contrôle. D'autres problématiques devraient être considérées dans le futur pour établir un modèle plus générique : la diffusion

des capteurs et actionneurs (répartis sur un réseaux local, voire accessible via internet), le contrôle de flotilles de robots coopérants, la gestion de multiples opérateurs humains dans la boucle de contrôle (e.g. collaboration à distance de plusieurs opérateurs humains), sont des exemples de problèmes qui n'ont pas été abordés dans ce modèle d'organisation.

La plate-forme de développement proposée au sein de la méthodologie CoSARC est celle que nous construisons autour du langage à composants CoSARC. Ce langage permet de modéliser et à terme de programmer et de déployer des architectures logicielles de contrôleurs, décrites à base de composants logiciels réutilisables. Il reprend et unifie certaines des propositions issues des langages de description d'architectures et des modèles de composants logiciels. D'abord, il intègre les notions de port et d'interface, qui permettent de favoriser un développement indépendant des composants logiciels. Il repose sur une séparation de différents aspects, chacun traité et décrit via différentes catégories de composants. Les composants de représentation décrivent les connaissances que possède le robot sur le monde dans lequel il évolue. Les composants de contrôle représentent les différentes activités constituant le système de prise de décision du robot. Les composants connecteurs représentent des protocoles d'interaction entre les composants de contrôle. Les configurations correspondent à des (sous-)architectures déployées sur des infrastructures matérielles. Le choix des différents aspects traités a été fait en fonction des besoins inhérents au développement de contrôleurs que nous avons identifiés. Pour chaque aspect à décrire, des notations adaptées sont proposées par le langage. La description du comportement des composants de représentation se fait via l'utilisation d'un langage procédural classique. La description du déploiement se fait via l'utilisation d'une notation propre et empirique, à partir de laquelle des outils informatiques spécialisés pouront à terme réaliser le déploiement réel, en considérant notamment son ordonnancement (aspect essentiel pour un système temps-réel). La description et la programmation des comportements asynchrones des composants de contrôle et de leurs protocoles d'interaction (connecteurs) sont faites par le biais du formalisme des réseaux de Petri à Objets. L'utilisation de ce formalisme de description amène des possibilités d'analyse des comportements des composants de contrôle et du modèle résultant de leur assemblage. Un aspect innovant du langage est donc sa spécialisation aux problématiques de modélisation posées par le domaine traité. Un autre aspect innovant est le mécanisme d'assemblage basé sur les composants connecteurs, qui permet à l'utilisateur de définir et de réutiliser les protocoles d'interaction et qui permet d'adapter les protocoles utilisés pour assembler des composants de contrôle.

L'utilisation du langage CoSARC a pour but de rendre plus rapide la production de code et l'entrée en phase de test, sans pour autant sacrifier complètement des étapes de formalisation. Notre volonté est d'utiliser une notation cohérente tout au long du cycle de vie d'une architecture logicielle, en appliquant des raffinements successifs simples à réaliser. Le mécanisme de raffinement du langage permet de gérer des composants à des niveaux de détails différents, en fonction de la phase du processus de développement dans laquelle ce composant est manipulé. Le raffinement permet en particulier de maintenir l'existence et la cohérence d'une architecture logicielle, depuis la phase de conception jusqu'à celle d'exécution, en passant par les phases de programmation, de description du déploiement. Pour que ce mécanisme de raffinement permette de rendre plus rapide et fiable le développement de contrôleurs, le développement de l'environnement du langage CoSARC est essentiel. Nous travaillons à l'heure actuelle à la réalisation d'un premier prototype de l'environnement d'exécution et nous commençons la production de l'environnement d'édition.

L'exemple de modélisation d'une architecture de contrôle d'un robot manipulateur mobile nous a permis de réaliser une première évaluation des qualités et des faiblesses de la proposition. La faiblesse principale est que le langage CoSARC reste compliqué à aborder pour des personnes ne maîtrisant pas la modélisation par RdPO et le modèle d'assemblage à l'aide de connecteurs. En effet, l'exemple traité a montré que certains modèles RdPO pouvaient atteindre une complexité qui les rend difficilement compréhensibles à la première lecture. Les RdPO des connecteurs ont également posé problème étant donné la nature dynamique de leur structure. Une des forces de

la proposition est que le développement par composants a permis de travailler sur des RdPO plus petits, car décrivant des comportements asynchrones de composants de granularité plus fine. Ceci a contribué à limiter la taille des RdPO de chaque composant. Une autre force du langage est de rendre la description des connecteurs indépendante de celle des composants de contrôle. L'apport en terme de modélisation est qu'on exprime explicitement les protocoles utilisés pour faire interagir les composants de contrôle, ce qui globalement permet de mieux comprendre la façon dont est organisée une architecture logicielle. Enfin, cet exemple a permis d'évaluer la façon dont le développement par composants pouvait permettre de construire des architectures de façon incrémentale en ajoutant ou en enrichissant des ressources. Le but à court terme est de programmer et de tester la ressource mobile, en partant du modèle présenté dans ce mémoire. Ensuite, le développement se poursuivra par intégration de nouvelles ressources et de nouveaux modes pour les ressources, en commençant par le manipulateur.

Le résultat de ces premières expérimentations sera déterminant afin d'évaluer vers quelles directions de recherche doit s'orienter notre travail afin de rendre plus facile la description des comportements et interactions des composants de contrôle. Dans ce sens, une des évolutions déjà envisagée consiste à hiérarchiser les RdPO. Dans un RdPO hiérarchisé, un sous réseau de Petri peut être encapsulé dans un macro-état et des transitions dites transitions d'exception permettent de sortir du macro-état (en fonction de l'état interne du macro-état ou d'un délai temporel). Ceci permettrait en particulier de modéliser et maîtriser plus facilement qu'avec un modèle non-hiérarchisé, l'interruption de séquences de calcul ou de situations d'exception. Néanmoins, l'ajout de la hiérarchisation aux RdPO demande une certaine prudence, car nous voulons garder la possibilité d'analyse des RdPO (la reconstitution d'un modèle RdPO non-hiérarchisé est donc essentielle). D'autre part, cela amène également la complexité supplémentaire de faire évoluer l'environnement d'exécution du langage CoSARC (et en particulier le pousse-jetons). Plus largement, l'enrichissement des notations manipulées dans le langage est un axe de recherche fondamental pour décrire de façon plus simple des comportements complexes. Une piste serait de décrire un ensemble de contraintes comportementales qui expriment des règles qu'un composant ne doit pas violer lors de son exécution. En effet, si les RdPO semblent adaptés à la description du fonctionnement "normal" d'un composant, leur utilisation devient laborieuse dès lors que l'on veut modéliser l'ensemble des situations "anormales" que le composant doit éviter. L'utilisation de contraintes pour exprimer ces situations pourrait, par exemple, servir à vérifier la validité des compositions. Une telle capacité de vérification permettrait de détecter des erreurs pendant les phases de conception et programmation. Une perspective à long terme est donc de définir une notation qui permette de décrire à la fois le comportement asynchrone et parallèle "normal" à exécuter et les situations "anormales" à éviter.

Les moyens de gestion du déploiement offerts par le langage CoSARC sont à l'heure actuelle insuffisants à la description complète d'un système temps-réel. Par exemple, l'ordonnancement temps-réel de bas niveau (des conteneurs) ne se base que sur les priorités d'exécution. Il est envisageable que d'autres propriétés, et éventuellement d'autres stratégies d'ordonnancement puissent être utilisées. Le but est de laisser aux développeurs les moyens de contrôler de façon simple et précise, la façon dont les conteneurs sont exécutés, en fonction des besoins applicatifs. Une des pistes à ce travail est d'enrichir le concept de conteneur afin qu'il soit vu comme un service technique, ou une composition de services techniques. Il est par exemple possible d'envisager différents services techniques disponibles sur un gestionnaire de système pour mettre en œuvre différentes stratégies d'ordonnancement. Le déploiement impose la prise en compte de contraintes de déploiement liées au logiciel et au matériel. Par exemple, certains composants de contrôle sont chargés de piloter des "périphériques" (capteurs, actionneurs, moyens de communication, composants électroniques dédiés). Ils requièrent la présence de matériels spécifiques sur l'hôte où ils sont déployés, ainsi que de logiciels spécifiques (e.g. drivers de ces périphériques, protocoles de communication) accessibles à travers le gestionnaire de système de l'hôte. Le respect de ces contraintes devrait pouvoir être vérifié de façon automatisée. Pour

cela, il est intéressant d'étudier la notion de *contrat de déploiement* pour les composants. Chaque composant pourrait posséder un ou plusieurs contrats de déploiement qui décrivent ses besoins en terme de services techniques offerts par l'infrastructure de déploiement. Un tel contrat exprimerait par exemple un ensemble de contraintes quant à la présence d'un matériel spécifique avec lequel un composant pourrait interagir. Cela nécessiterait donc qu'un gestionnaire de système CoSARC puisse fournir aux composants qu'il héberge un ensemble de services techniques liés à l'utilisation du matériel embarqué. Des propositions comme Concerto [43] ou RAJE [107] semblent particulièrement prégnantes pour répondre au problème général de description et de gestion de contraintes de déploiement, liées à l'utilisation de services techniques. La représentation explicite des services techniques pourrait permettre de décrire une infrastructure de déploiement en terme de compositions de composants techniques. Ces composants techniques permettraient par exemple de représenter des éléments matériels de l'infrastructure de déploiement (noeuds, liens de communication réseau, périphériques, etc.) que le niveau applicatif peut utiliser pour fonctionner. Ils permettraient également de représenter des services systèmes (e.g. création de conteneur, ordonnancement). Enfin, ils permettraient de représenter des services applicatifs externes (e.g. accès à un système de gestion de base de données, accès à une IHM, etc.). Intuitivement, ces composants techniques devraient posséder des ports leur permettant de fournir et de requérir des services techniques, eux-mêmes décrits par des *contrats de déploiement*. Une nouvelle catégorie de composants pourrait ainsi naître. Les composants seraient vus soit comme des abstractions représentant des parties matérielles/systèmes d'un contrôleur (gestionnaire de système, conteneurs, périphériques E/S, lien réseaux, capteurs, actionneurs, composant électronique, etc.), soit comme des parties logicielles de niveau applicatif (composant de représentation, connecteur, composant de contrôle), soit comme leur composition (une configuration déployée).

Le déploiement est une phase particulièrement critique, pendant laquelle le logiciel et le matériel du contrôleur sont couplés et le paramétrage système réalisé. La méthodologie CoSARC pourrait se diriger vers une conception, au moins partielle, de l'électronique et du logiciel des contrôleurs de robot. Plus généralement, une approche basée sur le Co-Design matériel/logiciel est certainement le futur de toutes les méthodologies de développement de contrôleurs de robots. En effet, la partie logicielle d'un contrôleur a souvent besoin d'interopérer avec des composants électroniques de traitements spécialisés (e.g. traitement du signal vidéo effectué sur FPGA). L'utilisation de tels composants est issue d'un besoin d'optimisation, dans le but de répondre à des contraintes de réactivité forte : les calculs les plus coûteux en temps sont déportés vers des composants électroniques avec lesquels la partie logicielle interagit. C'est donc une partie des calculs "métier" qui est déportée vers l'électronique embarquée. Une des pistes de recherche à adresser autour de notre projet est de considérer un composant indifféremment de son implantation logicielle ou matérielle pendant les phases de conception et de programmation : une même notation de modélisation est utilisée. Optimalement, ce n'est qu'au moment du déploiement qu'est prise la décision de traduire le composant sous une forme logicielle ou sous une forme matérielle. La gestion de la compilation des composants sous forme logicielle ou matérielle au moment du déploiement, permet de gérer au plus tard l'optimisation en fonction d'une application donnée. Pour cela, il doit exister des mécanismes de compilation différents qui s'appliquent à des contextes matériels d'exécution différents. Par exemple, lorsqu'un composant est déployé sur une carte PC embarquée, il prendra une forme purement logicielle. Pour une compilation vers une forme hybride logicielle/matérielle, l'utilisation de System On Chip (SoC) semble être une piste intéressante à étudier. Le coeur de processeur du SoC assure une partie de l'exécution d'un composant et de ses connecteurs (mémoire, communications) alors que, sur la partie "cablée" du SoC, le ou les circuits électroniques dédiés se chargent des calculs coûteux en temps. Une telle capacité de compilation revient à traduire le code des composants sous la forme de circuits et de composants électroniques. Pour cela, il est très intéressant de baser la compilation de certaines notations du langage CoSARC vers du code VHDL (code procédural, RdPO). En ce sens, notre équipe a déjà réalisé des travaux de compilation de réseaux de Petri vers du code VHDL, à travers le projet

HileCop [9], qui pourraient servir de base à cette évolution du langage CoSARC.

Au delà de considérations techniques d'évolution du langage CoSARC, les perspectives du travail réalisé au cours de la thèse, adressent également des problématiques purement conceptuelles. La définition de modèles d'organisation d'architectures de contrôle indépendants de toute plate-forme, est une piste à suivre à long terme. C'est à partir de ce modèle qu'est réalisée le "découpage" d'une architecture en un ensmeble de composants assemblés. Ce découpage logique influence les services fournis et utilisés par les composants (et donc leurs ports et interfaces), ce qui conditionne une grande partie de leur réutilisabilité. La production généralisée de modèles d'organisations, génériques ou spécifiques à un domaine d'application particulier (e.g. robotique sous-marine, productique), est un vecteur via lequel pourrait émerger un ou plusieurs modèles d'organisation standards. Notre équipe veut, dans cette optique, éprouver le modèle d'organisation proposé dans ce mémoire, en fonction de différentes applications robotiques étudiées au sein du laboratoire (e.g. robotique sous-marine d'exploration, robotique terrestre d'intervention, robotique chirurgicale, etc.), afin d'en déduire les lacunes et imprécisions. Le développement d'un ou plusieurs modes de coopération pour une flotille de robots mobiles permettrait d'évaluer, d'enrichir et de faire évoluer le modèle d'organisation proposé. De la même manière, la prise en compte de plusieurs plate-formes de télé-opération amènerait, là encore, à faire évoluer le modèle d'organisation initial. De façon plus globale, nous souhaitons nous inspirer des architectures comportementales proposées afin d'en déduire un modèle d'organisation générique. L'idée globale est d'essayer, à partir de différents modèles d'organisation, de proposer un modèle conjuguant au mieux les avantages des architectures délibératives et des architectures comportementales, et suffisamment abstrait pour s'appliquer à différents domaines de la robotique.

L'intégration de la méthodologie CoSARC au sein d'une démarche de développement dirigée par les modèles (MDA) permettrait de mieux quantifier l'apport que peux avoir une telle démarche sur le rendement d'un projet de développement. Un modèle d'organisation d'architectures logicielles de contrôleur, tel que celui que nous avons proposé, pourrait correspondre à un meta-modèle indépendant des plate-formes (meta-modèle PIM). Le meta-modèle du langage à composants CoSARC, correspond à un meta-modèle dépendant d'une plate-forme (meta-modèle PSM). Intégrer au sein d'une démarche de développement MDA les deux meta-modèles (PIM et PSM) nous semble intéressant pour plusieurs raisons. La première est un gain de temps de développement pour passer de modèles ébauches, décrits lors de la phase d'analyse, vers des modèles de conception écrits dans le langage CoSARC. Ce gain de temps pourrait favoriser la comparaison des différentes solutions technologiques (dont le langage CoSARC fait partie) pour le développement d'un même modèle d'organisation. La deuxième raison est la possibilité de réutiliser des organisations d'architectures (décrites en modèles PIM) en les transformant en architectures logicielles pour différentes plate-formes, au fur et à mesure de l'évolution technologique. Ce dernier point est d'autant plus intéressant dans le cadre de notre projet de recherche, que le langage CoSARC est voué à évoluer rapidement.

Références Bibliographiques

[1] R. Alami, R. Chatila, S. Fleury, M. Ghallab, and F. Ingrand. An architecture for autonomy. *International Journal of Robotics Research, vol.17, nř4 p.315-337*, 1998.

[2] J.S. Albus and al. 4d/rdc : A reference model architecture for unmanned vehicle systems. Technical report, NISTIR 6910, 2002.

[3] J.S. Albus, R. Lumia, J. Fiala, and A. Wavening. Nasrem : The nasa/nbs standard reference model for telerobot control system architecture. Technical report, Robot System Division, National Institute of Standards and Technologies, 1997.

[4] J. Aldrich, C. Chambers, and D. Notkin. Archjava : Connecting software architecture to implementation. In *24th International Conference on Software Engineering (ICSE'02), p. 187 - 197, ISBN :1-58113-472-X*, May 19 - 25 2002.

[5] J. Aldrich, V. Sazawal, C. Chambers, and D. Notkin. Language support for connector abstraction. In *Proceedings of ECOOP'2003, pp.74-102*, 2003.

[6] R. Allen. *A formal Approach to Software Architecture*. PhD thesis, Canergie Mellon University, May 1997.

[7] D. Andreu. Du modèle à l'exécution : proposition d'une architecture expérimentale ouverte. In *3èmes journées nationales de la recherche en robotique (JNRR)*, 15-17 Octobre 2001.

[8] D. Andreu, P. Fraisse, and A. Ségovia de los Rios. Teleoperations over an ip network : from control to architectural considerations. In *Eighth International Conference on Control, Automation, Robotics and Vision (ICARCV 2004), pp. 765-770*, december 6-9 2004.

[9] D. Andreu, T. Gil, and A. Nketsa. Implantation matérielle de systèmes complexes : Traduction automatique d'un réseau de petri non autonome en composants vhdl. *Journal Européen des Systèmes Automatisés, Hermès*, à paraître en 2006.

[10] R.C. Arkin. *Toward Cosmopolitan Robots : Intelligent Navigation in Extended Man-made environement.* PhD thesis, University of Massachusetts, Department of Computer and Information Science, COINS Technical Report 87-80, 1987.

[11] R.C. Arkin and T. Balch. Aura : principles and practice in review. Technical report, College of Computing, Georgia Institute of Technology, 1997.

[12] M. Augeraud and F. Bertrand. Objets à contrôle réactif. *Techniques et Sciences Informatiques*, 1999.

[13] J. Ayers. A reactive ambulatory robot architecture for operation in current and surge. In *Proceedings of the Autonomous Vehicle in Mine Countermeasures Symposium. Naval Postgraduate School, pp.15-31*, 1995.

[14] D. Balek. *Connectors in Software Architectures.* PhD thesis, Charles University, Faculty of Mathématics and Physics, Department of Software engineering, Malostranske namesti 25, 118 00 Prague 1, 2001.

[15] M. Barbier, J-F. Gabard, D. Vizcaino, and O. Bonnet-Torrès. Procosa : a software package for autonomous system supervision. In *First National Workshop on Control Architectures of Robots : software approaches and issues, pp. 37-47*, Avril 6-7 2006.

[16] R. Bastide. Objets coopératifs : Un formalisme pour la modélisation des systèmes consurrents. In *PhD thesis, Université Toulouse III*, 1992.

[17] R. Bastide, O. Sy, and P. Palanque. Formal spécification and prototyping of corba systems. In *13th European Conference on Object Oriented Programming (ECOOP99), Rachid Guerraoui, Volume Editor. Lecture Note in Computer Science nř1628, Springer (1999) 474-94*, 1999.

[18] R. Bastide, O. Sy, P. Palanque, and D. Navarre. Formal specification of corba services : experience and lessons learned. In *ACM Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA'2000), pp 105-117*, October 2000.

[19] K. Beck and al. Manifesto for agile software development, http ://agilemanifesto.org/.

[20] L. Berger. Interaction et modèles de programmation - support des interaction par les modèles à objets et à composants. *RSTI - L'objets. Coopération et Systèmes à Objets pp. 9-38*, 2002.

[21] G. Berry and G. Gonthier. The esterel synchronous programming language : Design, semantics, implementation. *Science of Computer Programming*, 19(2) :87–152, 1992.

[22] F. Bertrand and M. Augeraud. Control of asynchronous reactive objects. In *International Conference on data and knowledge Systems for Manufacturing and Engineering, pp. 539-544*, may 1994.

[23] A. Beugnard and R. Ogor. Encapsulation of protocols and services in medium components to build distributed applications. In *Engineering Distributed Objects (EDO '99), ICSE 99 Workshop*, May 17-18 1999.

[24] P. Binns, M. Engelhart, M. Jackson, and S. Vestal. Domain specific software for guidance, navigation and control. *International Journal of Software Engineering and Knowledge Engineering, vol. 6, no.2*, 1996.

[25] R.P. Bonnasso, D. Kortenkamp, D.P. Miller, and M. Slack. Experiences with an architecture for intelligent, reactive agents. Technical report, NASA Johnson Space center, 1995.

[26] G. Booch, J. Rumbaugh, and I. Jacobson. *UML User Guide*. Addison Wesley Publishing, 1999.

[27] J.J. Borrely, E. CosteManier, B. Espiau, K. Kapellos., R. Pissard-Gibollet, D. Simon, and N. Turro. The orccad architecture. *International Journal of Robotics Research, Special issues on Integrated Architectures for Robot Control and Programming, vol 17, no 4, p. 338-359*, April 1998.

[28] R.A. Brooks. A robust layered control system for a mobile robot. *IEEE journal of Robotics and Automation, vol. 2, no. 1, pp.14-23*, 1986.

[29] R.A. Brooks. A robot that walks : emergent behaviour from a carrefully evolved network. *Neural Computation, vol.1, no.2, pp.253-262*, 1989.

[30] R.A. Brooks, C. Breazeal, R. Irie, C.C. Kemp, M. Marjanovic, B. Scassellati, and M.M. Williamson. Alternative essences of intelligence. In *Proceedings of American Association of Artificial Intelligence (AAAI), p. 89-97*, July 1998.

[31] E. Bruneton, T. Coupaye, and J.B. Stefani. Recursive and dynamic software composition with sharing. In *Seventh International Workshop on Component-Oriented Programming (WCOP02) at ECOOP 2002*, Malaga, Espagne, Juin 2002.

[32] J. Bézivin. From objects to model transformation with the mda. In *Proceedings of TOOLS'USA, volume IEEE-TOOLS 39*, 2001.

[33] J. Bézivin and O. Gerbé. Towards a precise definition of the omg/mda framework. In *Proceedings of the conference on Autonomous Software Engineering (ASE'01)*, Novembre 2001.

[34] J.D. Carbou. *Conception d'une architecture pour la commande à distance d'un robot d'intervention*. PhD thesis, Université Montpellier II, France, 26 Janvier 2004.

[35] J.D. Carbou, D. Andreu, and P. Fraisse. Controle de robots autonomes basé sur les réseaux de petri hybrides. In *Conférence sur la modélisation des systèmes réactifs (MSR'01)*, 2001.

[36] J.D. Carbou, D. Andreu, and P. Fraisse. Events as a key of an autonomous robot controller. In *15th IFAC World Congress (IFAC b'02)*, 2002.

[37] E. Cariou and A. Beugnard. Specification of communication components in uml. In *The 2000 International Conference on Parallel and Distributed Processing Techniques and Applications (PDPTA'2000),editions CSREA, pp.785-791, vol.2*, June 26-29 2000.

[38] A. Carzaniga, A. Fuggetta, R.S. Hall, D. Heimbugner D., A. Van der Hoek, and A.L. Wolf. A characterization framework for software deployment technologies. Technical report, Technical report Cu-CS-857-98, Dept. of Computer Science, University of Colorado, 1998.

[39] F. Cassez and O. Roux. Compilation of the electre reactive language into finite transition systems. *Theoritical Computer Science*, 146, 1995.

[40] O. Causse and H.I. Christensen. Hierarchical control design based on petri net modeliing for an autonomous mobile robot. *Intelligent Autonomous Systems, U. Rembold Eds., IOS Press*, 1995.

[41] CLimbing and WAlking Robots (CLAWAR). http ://www.clawar.com/home.htm.

[42] Object Web Consortium. The fractal component model, http ://fractal.objectweb.org/specification/index.html.

[43] L. Courtrai, F. Guidec, and Y. Mahéo. Gestion de ressources pour composant parallèle adaptables. In *Journées Composant 2002*, 17-18 octobre 2002.

[44] A.A.D de Medeiros, R. CHatila, and S. Fleury. Specification and validation of a control architecture for autonomous mobile robots. In *IEEE International COnference on Intelligent Robots and Systems (IROS'96)*, November 1996.

[45] L.G. DeMichel, L.U. Yalcinalp, and S. Krishnan. Enterprise java beans specification v2.0. Technical report, Sun Microsystems, Aout 2001.

[46] S. Ducasse and T. Richner. Executable connectors : Towards reusable design elements. In *ESEC/FSE'97 (European Software Engineering Conference), LNCS (Lectures Notes in Computer Science), N 1301, pp. 483-500, Springer - Verlag*, 1997.

[47] B. Espiau. La peur des robots. *Science Revue , Hors série nř10, p.49*, mars 2003.

[48] B. Espiau, M. Ghallab, and F. Pierrot. Robotique domestique au japon. Technical report, Ambassade de France à Tokyo, Service pour la Science et la Technologie, rapport SMM04-097, 2004.

[49] B. Espiau, K. Kapellos, and M. Jourdan. Formal verification in robotics : Why and how ? In *International Symposium on Robotics Research*, October 1995.

[50] European Robotic Platform (EUROP). http ://www.robotics-platform.eu.com/missions.php.

[51] S. Fleury, M. Herrb, and R. Chatila. Genom : A tool for the spécification and the implmentation of operating modules in a distributed robot architecture. In *Internalional Conference on Intelligent Robots and Systems (IROS'97), vol.2, pp.842-848*, Septembre 1997.

[52] United Nations Economic Commission for Europe (UNECE) and International Federation of Robotics (IFR). World 2005 robotics, http ://www.ifrstat.org/.

[53] The Joint Architecture for Unmanned Systems (JAUS). Reference architecture specification, http ://www.jauswg.org/baseline/refarch.html.

[54] P. Fraisse, R. Zapata, W. Zarrad, and D. Andreu. Remote secure decentralized control strategy for mobile robots. *Journal of Advanced Robotics, pp. 1027-1040 (14)*, 2005.

[55] P. Freedman. Time, petri nets and robotics. *IEEE Transactions on Robotics and Automation, vol. 7, nř4, pp. 417-433*, 1991.

[56] National Science Fundation, NASA, NIBIB, and National Institue Of health. Wtec workshop : Review of u.s. research in robotics, http ://www.wtec.org/robotics/us-workshop/welcome.htm.

[57] E. Gama, R. Helm, R. Jonhson, J. Vlissides, and G. Booch. *Design Patterns : Element of Reusable Object-Oriented Software.* Addison-Wesley Professional Computing, 1995.

[58] D. Garlan, R. Monroe, and D. Wile. Acme : An architecture description interchange language. In *Proceedings of GASCON'97, pages 169-183*, Toronto, Ontario, USA, November 1997.

[59] E. Gat. On three-layer architectures. *A.I. and mobile robots, D. Korten Kamp & al. Eds. AAAI Press*, 1998.

[60] J.M. Geib, C. Gransart, and P. Merle. *CORBA : des concepts à la pratique.* Dunod, ISBN :2-10-004806-6, 1999.

[61] M. Gertz and P. Khosla. Onika : A multilevel human-machine interface for real-time sensor-based robotics systems. In *ASCE : SPACE 94 : The 4th Intternational Conference and Exposition on Engineering, Construction, and Operations in Space*, 1994.

[62] M.M. Gorlick and A. Quilici. Using weaves for software construction and analysis. In *Proceedings of the 13th International Conference on Software Engineering(ICSE13), pp. 23-34*, Austin, Texas, USA, May 1991.

[63] R. Grimes. *Professional DCOM Programming.* WRox Press ltd., Birmingham, Canada, 1997.

[64] Object Management Group. Omg model driven architecture, http ://www.omg.org/mda.

[65] Object Management Group. Uml ressource page, http ://www.uml.org/.

[66] Object Managment Group. Deployment and configuration of component-based distributed applications specification. OMG Draft Adopted Specifiation ptc/03-07-02, Object Management Group, June 2003.

[67] G. Hamilton. Java beans specification v.1.01. Technical report, Sun Microsystems, 1997.

[68] D. Harel. Statecharts : A visual formalism for complex systems. *The Science of Computer Programming, 8*, pages pp.231–274, 1987.

[69] D. Harel. Executable object modeling with statecharts. *IEEE COMPUTER, Vol. 30, issue 7*, pages pp.31–42, 1997.

[70] P. Hnetynka. Component model for unified deployment of distributed component-based software, citeseer.ist.psu.edu/719663.html.

[71] F. Ingrand. Architectures logicielles pour la robotique autonome. In *Journées Nationales de la recherche en Robotique JNRR'03*, Octobre 2003.

[72] ISO. Open distributed processing reference model. Technical Report ISO10746-1..4, International Standard Organization, 1995.

[73] J. Jacobson, G. Booch, and J. Rumbaugh. *Le processus unifié de développement logiciel.* Eyrolles, traduction française edition, ISBN2-212-09142-7, 2000.

[74] K. Jensen. *Coloured Petri Nets - Basic Concepts, Analysis Methods and Practical Use. ISBN : 3-540-60943-1, vol. 1, 2^{nd} edition.* Springer, 1997.

[75] Tahar Khammaci, Adel Smeda, and Mourad Oussalah. Active connectors for component-object based software architecture. In *Proceedings of the Sixteenth International Conference on Software Engineering and Knowledge Engineering (SEKE'2004), pp 346-349*, June 20-24 2004.

[76] G. Kiczales, J. Lamping, A. Mendhekar, C. Maeda, C. Lopes, J.M. Loingtier, and J. Irwin. Aspect oriented programming. In *11th European Conference on Object-Oriented Programming (ECOOP'97), volume 1241 de Lecture Notes in Computer Science, pp.220-242, Springer.*, juin 1997.

[77] C.A. Lakos. From coloured petri nets to object petri nets. In *16th international conference on Application and theory of Petri nets, Lecture notes in computer science 935*, pages 278–297, 1995.

[78] C. Lopes and W. Hursh. Separation of concerns. Technical report, College of Computer Science, Northeastern University, Boston, MA, Etat-Unis, Février 1995.

[79] D.C. Luckham and J. Vera. An event-based architecture definition language. *IEEE Transactions on SOftware Engineering, vol.21, no. 9, pages 717-734*, September 1995.

[80] J. Magee, N. Dulay, S. Eisenbach, and J. Kramer. Specifying distributed software architectures. In *Proceedings of the 5th European Software Engineering COnference (ESEC'95)*, Barcelone, Espagne, Septembre 1995.

[81] J. Magee, N. Dulay, and J. Kramer. A constructive development environment for parallel and distributed programs. In *International Workshop on Configurable Distributed Systems*, March 1994.

[82] J. Malenfant and S. Denier. Architecture réflexive pour le contrôle de robot autonomes. In *revue L'objet. Langage et Modèle à Objets LMO'04, pp.17-30*, Octobre 2004.

[83] R. Marvie. *Separation des preoccupations et méta-modélisation pour environnements de manipulation d'architectures logicielles à base de composants.* PhD thesis, Université des sciences et techniques de Lille, LIFL, UPRESA CNRS 8022 - Bat. M3 - UFR IEEA - 59655 Villeneuve d'Ascq Cedex, Decembre 2002.

[84] R. Marvie, P. Merle, J.M. Geib, and S. Leblanc. Openccm : une plateforme ouverte pour composants corba. In *Actes de la deuxième conférence française sur les Systèmes d'Exploitation (CFSE'2)*, pages pages 1–12, Paris, France, Avril 2001.

[85] N. Medvidovic, D.S. Rosemblum, and R.N. Taylor. A language and environment for architecture-based software development and evolution. In *Proceedings of the 21st International Conference on Software Engineering(ICSE'99)*, Los Angeles, California, USA, May 1999.

[86] N. Medvidovic and R.N. Taylor. A framework for classifying and comparing software architecture description languages. In Springer-Verlag, editor, *6th European Software Engineering Conference together with th 5th ACM SIGSOFT Symposium on the Foundations of Software Engineering (ESEC/FSE), pages 60-76*, pages 60–76, Zurich, switzerland, 1997.

[87] N. Muscettola, G.A. Dorais, C. Fry, R. Levinson, and C. Plaunt. Idea : Planning at the core of autonomous reactive agents. In *Proceedings of the 3rd International NASA Workshop on Planning and Scheduling for Space*, 2002.

[88] I. Nesnas, R. Volpe, T. Estlin, H. Das, R. Petras, and D. Mutz. Toward developping reusable software components for robotic applications. In *International Conference on Intelligent Robots and Systems*, October 2001.

[89] M. Nolar. Intégrer l'architecture d'une application avec son implémentation grâce à archjava. In *Mini-Workshop : Systèmes Coopératifs. Matière Approfondie*, 2002.

[90] OMG. Corba 3.0 new components chapters. OMG TC Document formal 2001-11-03, Object Management Group, Décembre 2001.

[91] OMG. Meta object facility v. 1.4. OMG TC Document formal 2002-04-03, Object Management Group, April 2002.

[92] M. Paludetto. Sur la commande de procédés industriels : une méthodologie basée objets et réseaux de petri. *PhD thesis, LAAS report nř91467*, december 1991, 10th.

[93] M. Paludetto and A. Benzina. Uml et les réseaux de petri, études de cas et évaluations temporelles. *Journal Européen des Systèmes Automatisés, Vol.36, Nř8,p.1155-1178. Rapport LAAS No02172.*, 2002.

[94] M. Paludetto and J. Delatour. Uml et les réseaux de petri : vers une sémantique des modèles dynamiques et une méthodologie de développement des systèmes temps-réel. *L'Objet, Vol. 5, p.443-467 . Rapport LAAS No99511.*, 1999.

[95] N. Pryce. Component interaction in distributed systems. In *4th International Conference on Configurable Distributed Systems*, Annapolis, Maryland, USA, May 4-6 1998.

[96] J.K. Rosenblatt. Damn : a distributed architecture for mobile navigation. Technical report, The Robotic Institute, Canergie Mellon University, 1997.

[97] D.C. Schmidt, D. Levine, and S. Mungee. The design of the tao real-time object request broquer. *Computer Communications, Elsivier Science, Volume 21, nř4*, April 1998.

[98] M. Shaw. Procedure calls are the assembly language of software interconnection : Connector deserve first-class status. In *Proceedings of Workshop on Studies of Software Design. Lecture Notes in Computer Science, SPringer-Verlag 1994*, May 1993.

[99] M. Shaw, R. DeLine, D. Klein, T. Ross, D. Toung, and G. Zelesnik. Abstraction for software architecture ans tools to support them. *IEEE Trans. Software Engineering, SE-21(4) :314-335*, Avril 1995.

[100] C. Sibertin-Blanc. Hign level petri nets with data structure. In *6th european workshop on Application and Theory of Petri Nets*, 1985.

[101] C. Sibertin-Blanc. Concurrency in cooperative objects. In *Proceedings of the Second International Workshop on High-Level Programming Models and Supportive Environments, HIPS'97, IEEE Socity Press*, Avril 1997.

[102] C. Sibertin-Blanc. Cooperative objects : Principles, use and implementation. *Concurrent Object Oriented Programming and Petri Nets, computer science XX*, 1998.

[103] R.G. Simmons. Structured control for autonomous robots. *IEEE Transactions on Robotics and Automation, vol. 22, nř1*, February 1994.

[104] D. Simon, B. Espiau, K. Kapellos, and R. Pissard-Gibollet. Orccad : Software engineering for real-time robotics, a technical insight. *Robotica, Special issues on Languages and Software in Robotics, vol 15, no 1, pp. 111-116*, March 1997.

[105] D. Simon, P. Freedman, and E. Castillo. Analyzing the temporal behavior of real-time closed-loop robotic tasks. In *International Conference on Robotics Automation, vol. 1, IEEE Computer Society Press, ISBN 0-8186-5330-2*, May 1994.

[106] A. Smeda, T. Khammaci, and M. Oussalah. Improving component-based software architecture by separating computations from interactions. In *First Internationnal Workshop on Coordination and Adaptation Software techniques for software entities (WCAT 04)*, June 14 2004.

[107] N. Le Sommer and F. Guidec. A contract-based approach of ressource constraint software deployment. In *First International IFIP/ACM Working Conference on component Deployment (CD 2002), nř2370 LNCS, Springer, pp.15-30, ISBN 3-540-43847-5*, June 2002.

[108] D.B. Stewart. The chimera methodology : Designing dynamically reconfigurable and reusable real-time software using port-based objects. *International Journal of Software Engineering and Knowledge Engineering, vol.6, nř2, pp.249-277*, June 1996.

[109] D.B. Stewart. Designing software components for real-time applications. In *2001 Embeded System Conference*, San Francisco, 2001.

[110] D.B. Stewart and P.K. Khosla. Rapid development of robotic applications using componant-based real-time software. In *IEEE/RSJ International Conference on Intelligent Robots and Systems (IROS '95)*, Pittsburg, USA, August 1995.

[111] D.B. Stewart, D.E. Schmidt, and P.K. Khosla. The chimera ii real-time operating system for advanced sensor-based robotic applications. *IEEE Transactions on Systems, Man, and Cybernetics, vol. 22, nř6, p.1282-1295*, November/December 1992.

[112] P. Stoetens. The complete orocos software guide : Open robot control software, http ://www.orocos.org/documentation.php.

[113] O. Sy. *Spécification comportementale de composants CORBA*. PhD thesis, Université de Toulouse 1, Toulouse, France, Mars 2001.

[114] C. Szyperski. *Component Software : Beyond Object Oriented Programming*. Addison-Wesley, 1999.

[115] S. Tai and S. Busse. Connectors for modeling objects relations in corba-based systems. In *24th International Conference on the Technology of Object Oriented Languages and Systems (TOOLS'24)*, 1997.

[116] F. Taïni, M. Paludetto, J. Delatour, and T. Cros. Vérification d'objets temps-réel à l'aide des réseaux de petri et de la logique linéaire. In *9th Conference on Real-time and Embedded Systems (RTS'2001), p-.65-77*, 6-8 March 2001.

[117] A. Tate, B. Drabble, and R. Kirby. O-plan2 : an open architecture for command, planning and control. Technical report, Artificial Intelligence Applications Institute, University of Edimburg, 1994.

[118] Rapide Design Team. Rapide 1.0 language reference manuals. Technical report, Program Analysis and Verification Group, Computer System Lab, Standford University, Juillet 1997.

[119] B. Thirion and L. Thiry. Concurrent programming for the control of hexapod walking. *ACM SIGAda, Ada Letters, Vol.22, Issue 1, p.17-28. ISBN :1094-3641.*, 2002.

[120] C. Urmson, R. Simmons, and I. Nesnas. A generic framework for robotic navigation. In *IEEE Aerospace Conference 2003*, March 2003.

[121] R. Valette. Petri nets for control and monitoring : specification, verification, implementation. In *Workshop Analysis and Design of Event-Driven Operations in Process Systems*, 10-11 April 1995.

[122] S. Vestal. Metah users manual, version 1.2. Technical report, Honeywell Technology Center Technical Report, 1998.

[123] R. Volpe, I. Nesnas, T. Estlin, D. Mutz, R. Petras, and H. Das. Claraty : Coupler layer architecture for robotic autonomy. Technical report, Jet Propulsion Laboratory, 2000.

[124] R. Volpe, I. Nesnas, T. Estlin, D. Mutz, R. Petras, and H. Das. The claraty architecture for robotic autonomy. In *IEEE Aerospace Conference (IAC-2001)*, March 2001.

[125] N. Wang, D. Schmidt, and D. Levine. Optimizing the ccm for high performance and real-time applications. In *Middleware'2000, Work-in-Progress session*, New-York, USA, 1999.

[126] R. Zapata, P. Lepinay, L. Torres, J. Droulez, and V. Creuze. Integration of biologically-plausible vision systems for controlling autonomous robots. In *SRA'00 : International Symposium on Robotics and Automation , pp. 503-508*, November 5-8 2000.

Travaux Publiés

Conférences francophones avec sélection

[cf1] **R. Passama**, D. Andreu, C. Dony et T. Libourel. Vers une méthodologie logicielle pour la robotique d'intervention, *Actes de la conférence ALCAA'04 (Agent Logiciels, Coopération, Apprentissage, Activités humaines)*, pp. 1-16, 2004, Montpellier, France.

[cf2] **R. Passama**, D. Andreu, C. Dony et T. Libourel. Modèle de contrôleur pour une méthodologie de conception basée composants, *Actes de la conférence 18ème JJCR (Journée des jeunes chercheurs en robotique)*, pp. 67-73, 2004, Douai, France.

[cf3] **R. Passama**. CoSARC : Une approche globale pour le développement de contrôleurs de robots, *Actes de la conférence JNRR'05 (5èmes Journées Nationales de la Recherche en Robotique), poster*, pp. 287-288, 5-7 octobre 2005, Guidel, Morbihan, France.

[cf4] **R. Passama**, D. Andreu, C. Dony et T. Libourel. Apports du génie logiciel pour la mise en oeuvre d'architectures de contrôle de robots, *Actes de CAL'06 (première conférence francophone sur les architectures logicielles)*, 9 pages, à paraitre, Septembre 2006, Nantes, France.

Conférences internationales avec sélection

[ci1] **R. Passama**, D. Andreu, C. Dony and T. Libourel. Control System Design using PNO-based Programmable Components, *Proceedings of CESA'03 : IEEE-IMACS International Conference on Computational Engineering in Systems Applications*, p. 6, 9-11 July 2003, Lille, France.

[ci2] **R. Passama**, D. Andreu, C. Dony et T. Libourel. Managing Control Architecture Design Process - Patterns, Components and Object Petri Nets in Use, *Proceedings of ICINCO 2006 (3rd IEEE International Conference on Informatics in Control, Automation & Robotics)*, 6 pages, à paraître, August 2006, Setubal, Portugal.

[ci3] **R. Passama**, D. Andreu, C. Dony and T. Libourel. Formalizing, Implementing and Reusing Controllers Behaviors and Interactions, *Proceedings of CESA'06 (IEEE-IMACS International Conference on Computational Engineering in Systems Applications)*, 8 pages, à paraître, october 2006, Beijin, China.

Autres

[a1] **R. Passama**, D. Andreu, F. Raclot et T. Libourel. J-Net Object : Un noyau d'exécution de Réseaux de Petri à Objets Temporels, *LIRMM internal Report nř02182 V.1.1, 2002*

[a2] **R. Passama**. Composants logiciels pour la conception de contrôleurs de robots, *Actes de la conférence Doctiss*, Juin 2005, Montpellier, France.

[a3] **R. Passama**, D. Andreu, C. Dony and T. Libourel. Overview of a new Robot Controller Development Methodology, *Actes de la conférence CAR'06 (1st conference on Control Architecture of robots)*, pp. 145-163, Avril 2006, Montpellier, France.

Troisième partie

Annexes

Table des figures

Glossaire

Actionneur : Périphérique permettant d'agir sur la partie opérative d'un robot.

Agrégation : Mécanisme de composition consistant à intégrer des assemblages de composants à l'interieur d'un composant, qui est alors qualifié de *composite*.

Approche asynchrone : Approche dans laquelle le temps est considéré comme continu, une action est préemptible et sa réalisation prend un temps non nul.

Approche synchrone : Approche dans laquelle le temps est considéré comme discret, une action est atomique et sa réalisation prend un temps nul.

Architecture de contrôle : Architecture logicielle d'un contrôleur de robot et l'infrastructure matérielle support de l'exécution, des communications et de l'accès aux entrées/sorties du contrôleur.

Architecture logicielle : Vue d'ensemble structurée d'une application logicielle, définie à partir de briques logicielles et d'interactions entre ces briques.

Assemblage : Mécanisme de composition consistant à connecter les ports des composants. Lorsque les ports d'un ensemble de composants sont connectés, on parle alors d'un assemblage de composants.

Asservissement : Application continue d'une loi de commande sur un robot.

Partie opérative : Partie mécanique et instrumentation embarquée du robot lui permettant d'interagir avec son environnement.

Capteur : Périphérique permettant d'observer l'état de l'environnement (capteur extéroceptif) ou de la structure du robot (capteur proprioceptif).

Commande événementielle : stratégie de commande d'un robot reposant sur un modèle discret.

Composant logiciel : Un composant logiciel est une entité logicielle déployable et exécutable, qui offre et requiert explicitement un ensemble de service à travers ses ports et qui peut être assemblé par connexion de ses ports.

Composition : Mécanisme permettant de mettre en relation des composants logiciels afin qu'il interopèrent.

Conteneur : Environnement d'exécution pour un ensemble de composants, qui fournit un accès et une utilisation simplifiéé à des services non-fonctionnels.

Contrôleur de robot : Partie logicielle chargée du contrôle de la partie opérative d'un robot et de la prise de décision, associé à un support physique d'exécution, de communication et d'accès aux entrées/sorties.

Cycle de vie d'un logiciel : Ensemble des étapes qui mènent de la conceptualisation à la mise en service d'un logiciel.

Déploiement : Phase du cyle de vie d'une application logicielle, qui consiste à diffuser, installer et paramétrer des composants sur les sites d'une infrastructure matérielle, afin de rendre l'application prête à être utilisée.

Infrastructure matérielle : Ensemble structuré de sites (i.e. nœuds de calcul) et de liens de communication inter-sites, sur lequel est déployée et exécutée une application logicielle.

Interface : Contrat définissant l'offre ou l'utilisation d'un service.

Interaction asynchrone : Interaction entre composants pendant laquelle le composant qui envoie un message ne bloque pas son activité en attente de la terminaison de l'activité du composant qui a reçu le message.

Interaction synchrone : Interaction entre composants pendant laquelle le composant qui envoie un message bloque son activité en attente de la terminaison de l'activité du composant qui a reçu le message.

Loi de Commande : stratégie de commande d'un robot reposant sur un modèle continu.

Mode : Fonctionnement particulier d'une ressource selon la nature de la relation de cette ressource avec d'autres intervenants (aucun, opérateurs ou robots).

Port : Point de connexion d'un composant associé à une interface, via lequel le composant offre ou utilise un service.

Ressource Robotique : Abstraction d'un sous-ensemble d'un robot, incluant un sous-ensemble de son contrôleur et un sous-ensemble de sa partie opérative.

Service : Fonctionnalité mise en œuvre par un ou plusieurs composants et utilisée par un ensemble de composants.

Stratégie de commande : stratégie définissant le contrôle des actionneurs d'un robot en fonction de données provenant de ses capteurs et d'une consigne.

Véhicule holonome : véhicule (tesrrestre) possédant 3 degrés de liberté sur un plan : 2 en translation (avant/arrière et gauche/droite) et un en rotation.

Véhicule non holonome : véhicule (tesrrestre) possédant 2 degrés de liberté sur un plan : 1 en translation (avant/arrière) et un en rotation.