A Modeling Language for Communicating Architectural Solutions in the Domain of Robot Control

Robin Passama

OBASCO Group, École des Mines de Nantes – INRIA, LINA
4 rue Alfred Kastler, 44307 Nantes cedex 3, France
`Robin.Passama@emn.fr`

Abstract. An open issue in robotics is to explain and compare very different solutions for control decomposition proposed by robot architects. This paper presents a domain specific modelling language dedicated to overcome this problem. The underlying goal of this work is to promote the communication of robot architects' expertise. The paper starts from a reflexion on the state of the art in research on control architectures. It results from this the report that to compare control architectures, it is necessary to be focused on the decision process organization by abstracting from technological features. A conceptual model for robotic architectures, defining domain terminology and concepts used in the description of decision process organization, is presented. Then, the language used to model organization of the decision process is defined via a meta-model. Finally, the use of the language is illustrated with the description of architectural solution of Aura.
keywords: robot control architecture, domain specific modelling language, decision process decomposition

1. Introduction

According to [10], a robot is "a machine that physically interacts with its environment to reach an objective assigned to it. It is a polyvalent entity able to adapt itself to variations of its working conditions. It has perception, decision and action functionalities [...] It also has, at various level, the ability to cooperate with man." A robot can be decomposed into two part (1) an operative - or mechanical - part including physical elements, sensors, actuators, etc, (2) a controller part in which decision are taken and reactions computed. The controller part is a complex composition of software and hardware pieces putting in place robot decisional process.

The design of controller is a very complex task. One important factor of this complexity is that design process involves a great variety of expertises in automatics, informatics, telecommunications, electronics, mechanics, etc. As intelligent robotics becomes an industrial challenge, the need of powerful methodologies, languages and tools now arises as a very important issue to reduce the complexity of the design of robot controllers. In this frame, the notion of *architecture*, as an artifact that describes a modular decomposition and main properties of the controller of a given robot, is certainly becoming as important as it is today in software engineering. An *architectural solution for robot control*, or *control architecture solution*, is a solution for decomposing design of controllers in a more or less generic way: the same solution can be used and specialized for the design of more than one robot controller, in a more or less restrictive way according to a given set of operational requirements associated to this solution.

To date, expressing architectural solutions is always made "by hand" in most of research or industrial papers, without formalization or standardisation, unlike it is done in software engineering with the used of UML [14]. Without a common language for designing robot control architectures , it so very difficult to understand, to communicate and to compare the different solutions. It is also a barrier for the adoption of robot control design patterns as solutions of specific robot control problems, like design pattern [12] are used in software engineering to express reusable software design solutions. UML could be used to design control architectures, but it does not support any domain abstractions, which leads each robot control designer to redefine new abstractions and terminology. The consequences would be that (1) concepts and terminology would not be compatible, making human understanding and solutions comparison very difficult, (2) users would reinvent the wheel for each of their design, wasting a considerable amount of time, (3) graphical representations of models would not emphasis the characteristics of domain expert "way of doing", making the communication of models more difficult. Furthermore, UML class and component diagrams, focus on software aspects. But a controller is an hybrid software/hardware (electronic) system and a same control architecture solution can be implemented in very different ways depending on the responsibilities granted to software and hardware parts.

That is why a domain specific modeling language for designing control architecture solutions is required. This paper presents a first attempt to define such a language. The difficulties to overcome are (1) the complexity and diversity of robot software controllers from both a structural and a behavioral point of view that makes it complex to find the right abstractions, (2) the difficulty to find the right separation of concerns in order to improve human understanding and expertise reuse, (3) the need to bridge the gap imposed by cultural and historical differences of practices in the community.

Section 2 presents a global reflexion on state of the art in control architecture design and make a synthesis of important concepts and practices. Section 3 starts from this synthesis and defines the proposed modeling language with a meta-model. Section 4 then illustrates the use of the language on Aura Architectural solution. Finally, section 5 concludes this paper and provides some perspectives.

2. Reflexion on the State of the Art in Robot Control Architectures

2.1. Challenge, Problem and Direction

Currently, the global challenge in robotics development is to express very different human expertise into identified software (and even hardware pieces) and to integrate all theses pieces in a cohesive manner into control architecture. This integration is not limited to expertise in robotic fields as control, navigation, vision, world modeling or artificial intelligence, but also concerns various "non-functional" domains as security, transactions, persistence and so on. Moreover, the emergence of service robotics will involve the necessity to express a large set of "business" domains, like medicine, human security, defense, etc. Managing each of independently already requires a high degree of expertise, but the emerging robotic industry will require managing them simultaneously for a given system. This challenge can be related to two complementary problems : *domain engineering* for the management of domain expertise and *separation of concerns* for expertise integration.

Historically, software engineering aims at providing solutions for software analysis, design, development, deployment and testing. But generally these solutions are centered on programmer's, software architect's or software project manager's points of view, like for example in object, aspect, or component paradigms. Even if software frameworks encapsulate specific domain expertises, they only provide a solution usable by programmers, but not by most of domain experts. Consequently, we see two main problems in the use of well-known software engineering approach: they don't provide to domain experts an adequate frame to apply their expertise to a given problem nor to reuse their solutions; most of them don't provide techniques at an adequate domain abstraction level to merge solutions from different domains of expertise into a global software system. In the frame of robotic systems design and development, this issue is a really important one since robot control architecture designers cannot all be software engineers.

The management of domain expertise at an adequate level of abstraction is the intended goals of the Domain Specific Languages (DSLs) approach [20]. The main idea of this pragmatic approach is to provide high-level languages to domain experts to describe solutions of domain problems. One advantage is that, thanks to the degree of abstraction of a DSL, empirical or formal rules can be checked on models in order to validate or to verify various domain properties. Another property of DSLs is sometime to allow for automatic code generation from solution models. The present paper investigates in that promising direction by defining the conceptual basis of a DSL for robot architects, but without considering analysis or code generation, only description of solutions being the subject of this work. In a DSL design, the first step, which is mandatory before being able to define language syntax and semantics that matches domain terminology, is the domain analysis. It consists to define common concepts to understand and communicate domain expertise. It corresponds, in our context, to the definition of common concepts usable by robot architects to explain control architecture solutions. To define domain concepts, next subsection gives a global overview of current methodological practices and their related issues.

2.2. Current Practices and Issues in Control Architectures

When studying current practices two main issues emerge: the diversity of control design methodologies [18] and the different levels of abstraction in the description of control architecture solutions.

This latter issue is discussed first. When looking at research papers on control architectures, one can notice that descriptions made are really different, uneasily comparable (if possible) and sometime quite "fuzzy". This report is certainly the first motivation of the present work. There are two factor for explaining this report. The first factor is the language or graphical conventions used to describe control architectures. Some authors use standard description languages like for instance UML class, component and deployment diagrams (for example [11]), which, as said earlier, limits the vision of the architecture to a specific implementation paradigm and does not really well captures the domain expertise. Many others use "ad-hoc" description features, which capture domain expertise at a greater abstraction level but which are often not well defined nor comparable. The second factor is certainly the merging of implementation detail with control decomposition details in the description, which impacts greatly in the diversity of terminologies and concepts. Many proposals are close to specific frameworks. For example CLARATy [29][28] is closed to a locomotion and navigation framework, LAAS [1] and ORCCAD [25] are close to specific execution frameworks and Chimera [27], OROCOS [24] and MirpaX [16] are close to communication frameworks. With so deep differences among proposals in their technological foundations, it is obviously difficult to denote recurrent (shared) concepts in all of them.

From this first report, the intuitive solution is to provide a domain language to allow the description of key design concepts without binding these concepts with underlying technologies, with implementation languages or paradigms or even with specific algorithms. The chosen direction should then be "abstraction", as in software engineering few years ago with the use of models written in a standard language [14] and of design patterns [12]. The direct consequence of this report is to differentiate the notion of robot control architecture from the notion of robot software architecture. As a first attempt to define these two notions, this paper proposes general definitions:

- The *robot control architecture* is a model of a robot controller that captures the decomposition of robot decisional process, its perception and action capabilities into interacting pieces of different levels of decisional complexity and of different responsibilities within each level.
- The *robot software architecture* is a model of the software system embedded in the robot, that defines the robot control architecture realization with software artifacts and interfaces it with robot hardware architecture.

So, regarding the previous conclusions, if the use of standard software modeling languages seems to be really useful to describe *robot software architectures* they are not adapted to describe *robot control architectures*. In the same time a language for designing *robot control architectures* is needed to avoid "ad-hoc" models and to allow for a better understanding, a better communication and a better comparison of robot architects' design solutions.

The other issue, regarding robot architects's practices, is the management of the diversity of control design methodologies. A classification of these methodologies, proposed in [18], has emerged along robotic history and defines four approaches: *reactive*, *deliberative*, *hybrid* and *behavior-based.*

Historically, the two main approaches for control architecture decomposition, defined since the 1980's, are deliberative and reactive approaches. The *deliberative* approach, also called *hierarchical* proposes a decomposition of a controller into a set of hierarchical layers, each one being a control and "decision-making system". A layer directly control the direct lower layer and is under control of its direct upper layer. The higher the layer is the more important are the decisions for the course of robot mission. The lower the layer is the more time constraints are strong in order to preserve robot reactivity (its ability to compute and apply adequate reactions in a time compatible with physical controlled system). Lower layers are responsible for simple control and reflex decisions like for example control law application loops or environment observation loops. Higher layers are involved in high-level and complex decisions such as planning. Classically, *deliberative* architectures are three layered [13], but some ones define a greater number of layers like *NASREM* [3] and *4D/RCS* [2]. Quickly speaking, the *deliberative* approach proposes a functional decomposition of robot decisional process from complex and long-term decision to simple and short-term ones. The main advantage of this approach is an interesting way to separate decisional concerns and the drawback can be a bad reactivity: the upper is the layer that makes the decision in response of a given stimuli, the higher the reaction time because information has to cross all lower layers to be handled. The *reactive* approach proposed by example in Brooks' subsumption architectures [7], proposes a decomposition of a controller into set of reactive autonomous entities, often called *reactive behaviors* because they implement behaviors of the robot specialized for a given finality (e.g. "reaching the nearest heat source"). Each reactive behavior define a *Perception - Decision - Reaction* cyclic process, where *Perception* is the mechanism that recovers sensor data, *Decision* is the mechanism that computes the adequate reaction and *Reaction* is the mechanism that apply to actuators the computed reaction. Interactions are not restricted to occur within a hierarchical schema: sensors data are simultaneously available to several reactive behaviors that decide of reactions and propagate them to actuators. The complexity is that many reactive behaviors can generate, at the same time, contradictory reactions. Reactive architecture so integrate an arbitration mechanism that allow the robot to adopt a coherent global behavior according to its mission objectives. Arbitration consists in recovering behaviors' reactions and synthesizing them in a global reaction that is actually applied to actuators. This arbitration is realized in different way, achieved by different types of interactions, for example with a complex vote protocol in *DAMN* [23] or with subsumption links in subsumption architectures [7]. The global behavior, issued from such a partially uncontrolled arbitration of behaviors is viewed as emergent. So, quickly speaking, the *reactive* approach proposes an multi-agent decomposition of robot decisional process. The main advantage is reactivity and adaptability according to physical world variations and the main drawback is an architectural and implementation complexity induced by the management of reactions arbitration.

To mix advantages of both approaches (understandability, manageability of architectures and reactivity of the resulting controller) the *hybrid* (for example *CLARATy*[29] or *LAAS* [1] architectures) and *behavioral-based* [19] approaches have been proposed. They are supposed to combine hierarchical decisional process decomposition and the ability to react quickly to environment stimuli. This is achieved in so many different ways that it is obviously complicated to list them all.

Nevertheless, a precise study of the domain shows that (1) each control architecture is based on a personal interpretation (by the authors) of the chosen approach and (2) that the distinction between all these approaches can be really fuzzy. First of all, the distinction between *deliberative* and *reactive* approaches, that seems to be clear can be attenuated by the fact that lower layer of deliberative architectures contain reactive behaviors (i.e. control law application loops) and by the fact that reactive architectures can be arranged according to a set of hierarchical layers representing different levels of decision complexity [7]. What primitively differentiate these architectures is the underlying decomposition "philosophy" (functional and multi-agent).

Another example is the difference between reactive and behavioral-based approaches that is really thin since it mainly relies on a criterion ("the behavior-based can store representations while reactive cannot" [18]) that can be considered as subjective (in fact, only the life time of the representation differs).

One more example is the difference between hybrid architectures like *Aura* [4], *3T* [5] or *ARM-GALS* [17] on the one hand and *LAAS*[1], *CLARATy* [29] on the other hand. In fact, the first ones are hybrid in the sense that their lower layer is organized around reactive behaviors coordinated with an arbitration mechanism (like reactive or behavior-based approaches) and this layer is under the control of a decisional layer in charge of complex decisions (resulting in reconfigurations of reactive layer). The second ones are more or less organized as deliberative architectures but according to the authors with a greater uncoupling between layers which is a quite subjective criterion. Fortunately this criterion can be specified thanks to architectures, for example *ORCCAD* [6] or *LIRMM* [22] architectures, that clarify in a more or less explicit way this point: architectures are viewed as hybrids because they have a layered style that allows for direct interactions (under given conditions) between non adjacent (not directly in relation) layers, allowing so sensing or reaction information to cross frontiers between layers in order to improve reactivity. To conclude on that point, the difference between hybrid architectures is important when considering the control design within the lower layers: the former approach being based on a multi-agent decomposition, the latter being rather based on a functional one.

The final example is the difference between hybrid architectures like *IDEA* [21] or *Chimera* [26] and other hybrid architectures, that lies in the decomposition of the control architecture into control sub-systems that incorporate the control of specific parts of the controlled robotic system. This organization is viewed as hybrid because (1) each subsystem encapsulates both reactive and deliberative capabilities and so can be viewed as a hierarchically layered architecture (even if it is not explicit in papers) and (2) subsystems are independent from each other and coordinate in a complex way, being so considered as autonomous interacting agents. This organization is, in a limited way, generalized in *LIRMM* [22] but also in *CLARATy* [29] where each subsystems is in turn incorporated into a more global layered system that controls the whole robot: subsystems are then under the control of a higher decisional layer. This approach has the benefit to render more intuitive the decisional process decomposition because it couples it with the decomposition of morphological and infrastructural (hardware) attributes of the controlled sub-system.

As a conclusion for this subsection, the intuitive direction to follow for a language for communicating architectural solutions is to provide common concepts that on one hand abstract from these subtle differences between approaches and on the other hand allow all existing control design methodologies to be used.

2.3. Commonalities

Defining the right abstractions to design control architecture solutions requires in the first time to focus on commonalities between proposed architectures. Commonalities are identified in different ways. The first (and rather classical in domain analysis) way is to consider as ”common” the concepts that are the most recurrent in (since there is no concept shared by all) control architectures. The second is to consider as ”common” the concepts that are explicitly used in really few architectures but that are in fact really useful and always applicable for control architecture decomposition. The last way consists in identifying recurrent variations in control architectures and generalizing them into a concept that can capture all possible variants.

The most easy to find recurrent concept is the one of *layer*. The *layer* concept exists in a huge number of architecture, initially in *deliberative* architectures, but also in most of *hybrid*, *reactive* and *behavior-based* architectures. Unfortunately, one have to admit that nearly each author has its own vision of what is exactly a *layer*. For example, in *subsumption architectures* [7] layers are abstractions of reactive behavior roles (e.g. robot movements, robot integrity, etc.), and in *LAAS architectures* [1] they separate AI decision-making mechanisms from control law modules. What is important in the concept of layers, is that a layered organization separates decisional concerns in a clear way, whatever the precise meanings of layers are. The layer is so the vector of a hierarchical decomposition of decisional process.

Another recurrent concept is the one of *activity* or *task*. An *activity* or *task* denotes a part of the decisional process with specific decisional, control and perception responsibilities. The terms *activity* or *task* are not standards since many other terms are often use, like for example real-time task and real-time procedure [6], reactive behavior [7], genom module [1], module [9], agent [21], port-based object [26], Motor or perception Schema [4], depending on authors point of view that is influenced by implementation or methodological concerns. Sometimes *activities* are not explicit in the architecture, even if they exists, for example *CLARATy* architecture hides control and perception threads inside its object-oriented design. In fact this latter point is also true for nearly all layered architectures, since upper ”decisional” or ”deliberative” layers are made of entities (often explicitly drawn) that are responsible of AI-based planning [1] [29] or environment representation and navigation [4] mechanisms. These entities can also be viewed as *activities* of higher level of decision. One important property is that *activities* are arranged into *layers* according to their decisional complexity.

As reported in the previous subsection, some architectures propose a decomposition of control architectures into *systems*. The concept of *system*, as a part of the decisional process that incorporates the control of specific parts of the controlled robot, explicitly exists in a really limited number of architectures and under different names and forms like IDEA agents [21] or Robotic Resource [22]. This concept also exists in other architectures but without being explicitly defined. The advantages of this concept are that (1) it is useful for decomposing decisional process (cf. previous subsection) and (2) it can be generalized in such a way that it is applicable for all architectures. The generalization consists in a *systemic* organization of control architectures: a system can potentially be decomposed into subsystems that are systems responsible of the control of sub-parts of the controlled physical part of robot. Sub-systems being systems they can in turn be decomposed in such a way. Since there can be a single system for decomposing the control architecture of a given robot (i.e. no systemic decomposition) and that an entire robot team can be viewed as a *system* itself decomposable into subsystems (one for each robot), the concept of *system* can be easily applied to all control architectures. One can notice that the *systemic* organization is orthogonal to the *hierarchical* one: each subsystem can incorporate both reactive and long-term decision-making *activities* and so can itself be *layered*.

Another useful but rather exceptional concept is the *knowledge* concept, that helps to identify the data and know-how used in a control architecture. If never mentioned as with the term *knowledge* this concept explicitly exists in architectures using object-oriented models in their description, more precisely *CLARATy* [29] and *LIRMM* [9] architectures. Object-class hierarchies represent the knowledge used in the architecture, for example robot physical properties knowledge or environment knowledge. The functional layer of *CLARATy* partially merges knowledge with activities while these two aspects are clearly separated respectively in object class and Petri-net modules in *LIRMM* proposal. The advantages of this concept are that (1) it is useful for qualifying more precisely the responsibilities of *activities* by explicitly defining what type of information they use and (2) it implicitly exists in all architectures. Indeed, whatever the *activity* taken into account in any architecture, it uses a specific knowledge of the robot and the environment. For example, any control *activity* is based on a representation of the robot morphology and kynodynamics. Of course, this representation is in most of time completely hidden in the architecture design. Another related aspect is *knowledge specialization*, effective in *CLARATy* thanks to class specialization mechanism. *Knowledge specialization* helps to define different levels of knowledge refinement and to qualify more precisely at which level of refinement an *activity* works.

One thing that emerges when studying the different architectural solutions proposed along history is the huge diversity of interactions used. Nearly each architecture use a specific set of interaction protocols, of different degrees of abstraction according to their implementation. Interactions are the most recurrent variations that can be found between control architectures. Since its impossible to list all possible interactions (more especially as new ones could be defined in the future), a concept generalizing them is necessary in a domain language for control architecture description. Interaction partly influence the control design methodology. For example, specific protocols, like subsumption links [7] or vote protocols [23], are used in *reactive* and *behavior-based* architectures to put in place arbitration mechanisms; layered architectures use different event notification protocols to make sensing activities communicate with upper layer activities [9]. If interactions can be explicitly represented, it would certainly be useful for a better understanding of the design methodology used for a given control architecture solution.

3. A Control Architecture Modeling Language

The proposed modeling language focus on the control *control architecture* -i.e. the decomposition of robot decisional process, and does not take into account implementation aspects -i.e. robot software and hardware architectures.

3.1. Concepts, Terminology and Graphical Conventions

One important thing to take into account when designing a domain specific language is that it has to be complete, minimalist and easily readable. Complete, because each domain solution has to be expressible with the language. Minimalist, because the minimal set of concepts has to be provided to express solutions. Easily readable because a quick understanding of a solution is necessary. This is a big challenge in itself because the right balance has to be chosen between a great variety of precise domain concepts on one hand and a restricted set of generic domain concepts that can be more easily learn on the other hand.

3.1.1. Main Concepts

First, we define four main concepts: *Knowledge*, *Activity*, *Coordination*, *System*. To this end, the term task is used, and it has to be understood in the general meaning, -i.e. the fact of doing something.

Knowledge: a *Knowledge* entity identifies a structured piece of information about the world within which the robot controller evolves. It can directly refer to the physical world (environment, robot body) or to a concept bound up to this physical world like a "phenomenon" or an "event". It can also refer to a know-how of the robot relative to this world, i.e. a way of detecting/solving problems relevant to this world (e.g. criteria defining singular configurations of the robot). Finally, the control architecture being itself part of the robot world, a *Knowledge* entity can also explicitly refer to (parts of) it, to get for instance a form of introspection.

Activity: an *Activity* entity is responsible for the achievement of a task that plays a role in the robot decisional process, using a set of *Knowledge* entities. For example, an *Activity* entity (activity for short) can refer to the observation of the environment state, of the robot (body) state, or even of the controller state. It can also refer to: low-level control of the robot, like the application of a control law, medium-level control as control context commutation, and high-level control like planning. Finally, it can also refer to a learning activity (creating or refining knowledge) whatever the level of control is concerned. An activity uses (internally) *Knowledge* entities to elaborate its decisions. For example, an activity defining a control law application uses knowledge on environment and robot morphology to compute actuator commands.

Coordination: a *Coordination* entity is responsible of the way a set of activities interact by exchanging or sharing knowledge. For example, it can express collaboration, i.e. an interaction type defining how different tasks are distributed among activities to achieve a more complex task. This is the most common case of *Coordination* entities, for instance command execution requesting or event notification. It can also express competition, i.e. an interaction type defining how several activities achieve the same task. This the the type used in many *behavior-based* or *reactive* architectures, for instance vote protocols or subsumption links. A same activity can be involved in more that one coordination. *Knowledge* entities that are exchanged or shared during coordination depends on the nature of the interaction. For example, in an

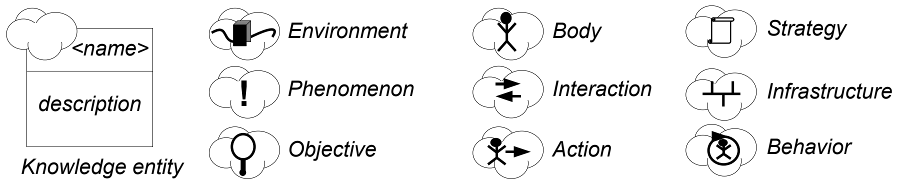


Figure 1. Some graphical conventions for representing *Knowledge* entities

event notification, *Knowledge* entities can represent specific states of the observed physical system (environment or robot body).

Systems: a *System* entity is an abstraction of the control of a physical (mechanical) entity (morphologically distributed or not). For example, a *System* entity (system for short) can refer to the control of a robot's part (e.g. the arm or the vehicle of a mobile robot), to the control of the (entire) robot (e.g. the mobile robot) or even to the control of a robot team. It is responsible of the way a set of *Activity* and *Coordination* entities, concerning this physical entity, are organized in order to achieve a set of tasks. This organization is done with a hierarchy of *layers*. A *layer* is an abstraction that symbolizes a "level of decision complexity" whatever the complexity of the decision is (from "simple" reactive decisions to "complex" deliberative decisions). A system organizes its internal activities by associating each of them to a *layer* according to its relative decision complexity in the system. A system also defines relations between its internal control architecture and *infrastructure* elements, like sensors and actuators. An *infrastructure* element is an abstraction that represents primitive part of the robot that provides to systems inputs (sensors), outputs (actuators) or both (physical communication links) or abstract composition of that parts (e.g. all actuators of the arm) by which the system retrieves information from the world within which the robot evolves.

3.1.2. Knowledge

A *Knowledge* entity is quite complex to detail without more specialization. At the highest level of abstraction, the only thing that can be said is that it contains a model synthesizing the type of knowledge and data representing the parameterization of this model for a given context. "Model" means any form of representation of the knowledge, being it mathematical formulas, empirical models, declarative models like CSP, biologically inspired models like neural networks, geometrical models, etc.

Since *Knowledge* entities are used to qualify other entities, its specialization to more precise domain abstractions is a way to improve easy understanding of models. In this way, the work done by Brugali and Salvaneschi in stable aspects of robot development [8] provides a good terminological basis that is extended here. So, knowledge entities can be arranged following three categories: *Embodiment*, *Situatedness* and *Intelligence*.

The *Embodiment* refers to the consciousness of having a body that allows the robot to experience the world directly. In this category, we find Knowledge entities like that representing the *Body* or *Infrastructure* of a robot, or some of their parts. The *Body* Knowledge entities allow for representing the electro-mechanical devices of robot operative part. *Body* entities can be themselves associated with other Knowledge entities of the *Embodiment* category. For instance a *Body* is associated to *Morphologies* and *Kinodynamics* [8] that are physical properties of robot body. A *Morphology* knowledge entity represents shape of a *Body*, its physical components and their structural relationship. For example, a robot can have a humanoid morphology, an animal-inspired or a human-vehicle inspired morphology. A *KinoDynamics* knowledge entity represents the kinematic (position and velocity) and dynamic (acceleration, force) constraints that limit the relative movement of the robot's body in the environment (morphological constraint like links and joints, physics law like gravity, etc.). The *Infrastructure* entities allow for representing the electronic-communications devices of robot controller part. *Infrastructure* entities represent elements like *Sensors* and *Actuators* or even more complex composite elements. *Body* and *Infrastructure* entities can themselves be decomposed into smaller part. For example, a rover robot entire *Body* can be decomposed into an *Arm*, a *Vehicle* and a *Camera*. Knowledge on embodiment can be so modularized.

The *Situatedness* refers to evolving in a complex, dynamic and unstructured environment that strongly affects the robot behavior. In this category, there are Knowledge entities like that representing the *Environment*. The environment is viewed as a continuum of physical configurations, but from a decisional point of view it is a discrete spatial-temporal milieu that is made up of any kind of dynamic or static elements such as people, robots, equipments, buildings, animals, mountains, etc., and hosts all their (potential) mutual *Interactions*. It can be decomposed into sets of *Environment* entities that represent any spatial part of it. *Interactions* are knowledge entities that represent interactions that can occur between the robot and the environment. Contrary to [8] the concept of *Interaction* is here limited to interactions governed by physic's laws (movement in environment, objects grasping, physical quantities measurement, etc.). In this category, there are also Knowledge entities like that representing the *Phenomena*. A *Phenomenon* refers to something that can occur in the environment, which is directly perceptible (thanks to specific *Interactions*) or estimable by the robot. To describe these entities we need others ones like those representing *Places* in the environment (the "where"), *Objects* (the "what") and *Time* (the "when").

The *Intelligence* refers to the ability of the robot to adopt adequate and useful behaviors while interacting with the dynamic environment. In this category, we can find *Behaviors* and *Actions*. Briefly, an *Action* denotes a capability of

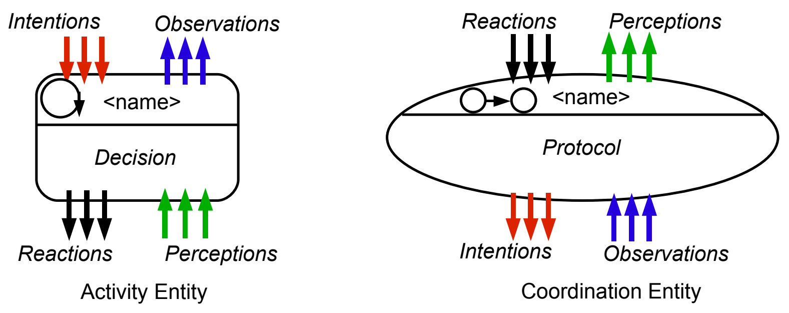


Figure 2. Graphical conventions for representing *Activity* and *Coordination* entities

the robot (or part of it) to act in order to obtain a given result (e.g. a given effect on the environment). For example, an *Action* can represent the fact of ”reaching a given place at a given time”. The expected result of *Actions* can be represented by *Objectives* (e.g. the place and time). *Behavior* denotes an abstract representation of the way the robot behaves when realizing *Actions* according to the possible robot-environment *Interactions*. For instance, a *Behavior* can be ”being attracted by the nearest heat sources”, it is defined thanks to *Actions* like ”reaching a given place”, ”detecting heat sources” and ”defining nearest reachable source” and according to *Interactions* like ”heat measurement”, ”movement in room” and ”environment perceiving”. *Knowledge* entities of this category can also represent *Strategies* associated to each *System*. A knowledge entity representing a *Strategy* contains an action planner and for each action defines the behavior(s) to be selected (and if necessary merges) to reach its objective. *Behavior* is the result of the activation of a set of coordinated *Activities* (or a single one). So, to allow the precise description of *Strategies*, Knowledge entities can also represent the *Activities* and *Coordination* used to give concrete expression to the *Strategy*, to allow the robot to reason on. This conceptual decomposition of *Intelligence* is partially detached from [8] proposal, to put in adequacy the organization of knowledge entities with other main domain concepts.

Figure 1 shows some graphical conventions used to describe *Knowledge* entities. The different types of *Knowledge* entities are represented using different symbols to clarify their intrinsic differences. To this point, concepts allows only to model the ”passive” characteristics of robot controller architecture, not the ”active” ones.

3.1.3. Activity and Coordination

”Active” characteristics are expressed thanks the *Activity* and the *Coordination* entities. An *Activity* is an entity of any level of decision that puts in place *Perception-Decision-Reaction* cycles. Graphical conventions to represent them and their properties are represented in figure 2.

It receives *Perceptions* (required pieces of information, or significant phenomena notification) from other activities or from the infrastructure (sensors). Each *Perception* of an activity is associated with one or more knowledge entities of any level of abstraction, from simple sensor data to complex computed robot or environment states. It contains a *Decision* mechanism that computes *Reactions*. This mechanism is of any level of abstraction, from simple control law computation to a high-level planning or supervisory control mechanism. The *Decision* mechanism handles activity internal knowledge (*Body* and *Environment* for instance) and knowledge coming from *Perceptions* to determine the adequate *Reactions* to adopt. *Reactions* represent the way an activity wants its *decision* to be realized by (eventually) others activities or by the infrastructure (actuators). Each *Reaction* is associated with one or more knowledge entities of any level of abstraction, from simple actuator data to a high-level order (e.g. an *Objective*). The *Decision* mechanism can be influenced by *Intentions* it receives. An *Intention* represents a goal that an activity intends to accomplish -i.e. a goal influencing its *Decision*. An *Intention* is associated to knowledge entities representing, for instance, the desired state of the robot Body, related or not to the *Environment*, or a desired *Behavior*. The *Reaction* emitted by an activity can be viewed as an *Intention* by the activity that receives it. The *Decision* mechanism can also emit *Observations*. An *Observation* represents an interesting state of: decisional process, body or environment. For instance, an activity that detects an obstacle in the environment can transmit the corresponding *Observation* associated with corresponding knowledge entities representing the obstacle. An emitted *Observation* can be viewed as a *Perception* by activities that receive it. Like for knowledge entities, activities could be categorized into more specific entities like for instance, *Environment Observers* or *Body Motion Planners*, and so on, but this specialization should take place at the moment when a consensus on decisional process decomposition will be accepted.

Intentions, *Observations*, *Reactions* and *Perceptions* are exchanged by activities thanks to *Coordination* entities. A *Coordination* entity is an entity of any level of abstraction, that imposes a protocol for knowledge exchange or share between a set of activities. It can represent various interactions like simple event notifications and resource request on time interval as the one used in CLARATy, specific subsumption or inhibition links, as well as a complex vote protocol like in DAMN. In fact, it depends on the protocol and on the nature of *Intentions*, *Observations*, *Perceptions* and *Reactions* taken into account by the *Coordination*. Graphical conventions to represent *Coordination* entities are presented in figure 2. *Intentions*, *Observations*, *Perceptions* and *Reactions* are optional features for both activity and coordination entities (even if using none of them makes no sense).

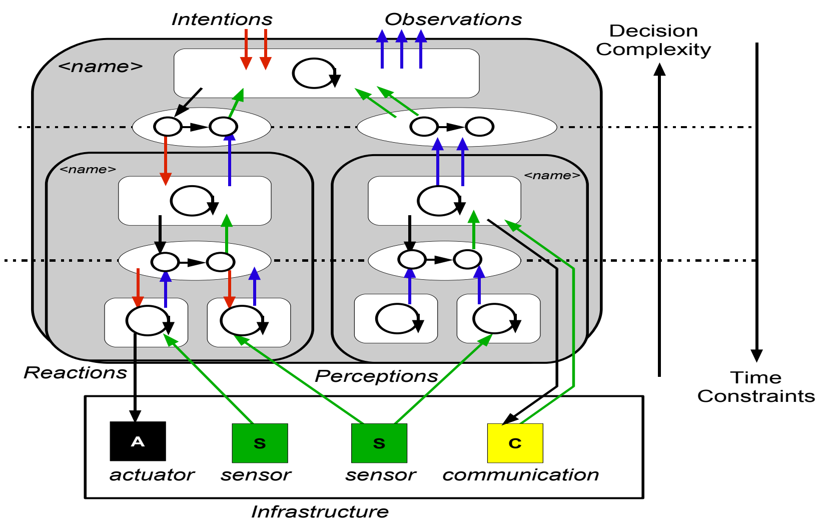


Figure 3. Graphical conventions for representing *Systems* entities

3.1.4. System

Systems are entities used to describe control architectures. Each *System* contains a set of coordinated activities and share with some of them some of their inputs and outputs (i.e. *Intentions*, *Observations*, *Perceptions* and *Reactions*). So a *System* can be considered itself as a decision-making system just as activities and can coordinate with other activities and/or *Systems*. Inside *Systems*, activities are organized according to a layered approach. The number and the precise semantics of each layer (decisional, reactive, executive, functional, etc.) is let undetermined to allow a maximal flexibility, main hierarchical organization criterion being the ”decisional complexity” (increasing from down to up) and real-time constraints (increasing from up to down).

Systems being used to describe the control architecture of an identified piece of the robot, they are in relation with *Body* Knowledge entities they are responsible of. These latter are useful to ”reason” about the robot body while systems are useful to exploit it. For example a *Manipulator System* control a robot *Arm*. System being able to contain other systems, the control of the robot or group of robots can also be described recursively. For example, the system controlling a rover robot can be composed of one system controlling its arm (*Manipulator System*) and another one controlling its vehicle (*Locomotor System*), like in CLARATy. All activities contained in a *System* participate to the control of the same part of the robot *Body*. A *System* can also explicitly refers to the *Infrastructure* of the *Robot Body* to associate its *Intentions*, *Observations*, *Perceptions* and *Reactions* with related *Infrastructure* elements, and more particularly *Sensors* and *Actuators*. For example, the *Manipulator System* refers to joints sensors and actuators of the arm and put them in relation with its *Perceptions* and *Reactions*. Graphical conventions for representing systems and robot infrastructure are presented in figure 3. This figure represents a *System* containing two *Subsystems*, where *Subsystems* contains two layers (layers are differentiated by dotted lines) and the *System* contains three layers, including the two preceding layers and an upper layer. Activities contained in *subsystems* interact with sensors and actuators of a more global infrastructural element (e.g. arm sensing and actuating infrastructure).

3.2. Language Meta-model

Now that concepts have been defined, this section presents the modeling language meta-model which reifies these concepts at the modelling level.

3.2.1. Reifying Concepts

The diagram of figure 4 shows the way main concepts are reified into the meta-model. Interesting things to denote in this diagram are:

- All control architecture entities are generalized into an *Entity* abstract class, in turn specialized into two abstract class *Knowledge entity* and *Decisional Entity*. This latter class is a generalization of all entities used in decisional process decomposition.
- A composite pattern is used to describe relations between *Activity*, *Coordination* and *System* entities allowing so ”recursive” decomposition of architecture into coarse-grained *Systems*.
- A *Decisional Entity* uses internally a set of *Knowledge Entities* (association with the *used* role).
- All types of inputs and outputs of *Decisional Entities* are generalized into a *Interaction Point* abstract class. According to the diagram, a same *Interaction Point* can be shared by more than one *Decisional entity*, for example

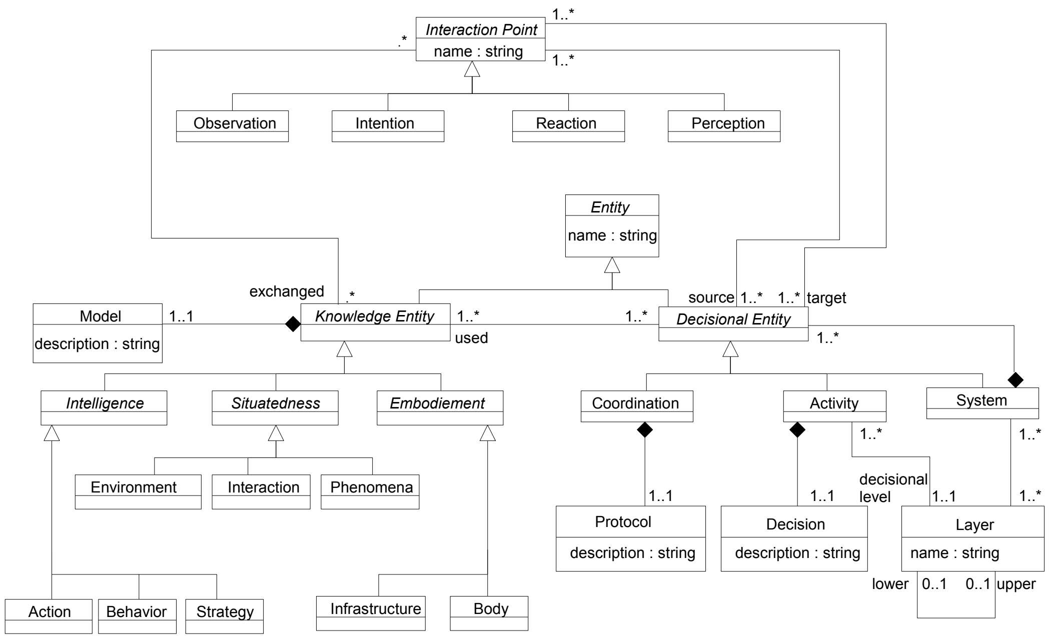


Figure 4. Meta-model : diagram of main concepts

a same *Reaction* can be shared by an *Activity* and its containing *System* (than in fact exports this *Reaction* outside its limits). In consequence it can have more than one source and more than one target.

- An *Interaction Point* exchanges a set of *Knowledge Entities* and a same *Knowledge Entity* can be exchanged by any number of *Interaction Points*.
- A *System* contains a set of hierarchically ordered *layers* (according to the *lower-upper* relation) and *layers* can be defined across many *Systems*.
- An *Activity* is associated to a unique *layer* that corresponds to its ”decisional level” and a *layer* can contain many *activities*.

The meta-model does not precise the way *Models*, *Protocols* and *Decision* are described. Their string *description* attribute is added to allow for a description in a natural language, which of course is not formal but, in a first time, the most simple solution is preferred.

This diagram shows that *Knowledge Entities* are shared by the other entities. This is explained by the fact that a same knowledge can be use in many parts of the robot control architecture. For example knowledge on robot body is used in all activities that put in place control loops. This can be compared to a kind of separation of concerns since *Knowledge Entities* can be viewed as aspects that crosscut the decisional process decomposition. This is taken into account in the language by differentiating knowledge representation concern from decisional decomposition one: *Knowledge Entities* are modeled in a first dimension and *Decisional Entities* in a second one. This two dimensions are related to each other with links (represented in decisional dimension) between *Knowledge* and *Decisional Entities* (relations where *Knowledge entities* play *used* and *exchanged* role), which represents aspects weaving.

3.2.2. Details on Knowledge Entities

Relations between the different types of *Knowledge Entities* are described in the diagram of figure 5. With the purpose of conciseness, the diagram does not precise all meta-classes, like *Time*, *Places*, *Objects*, *Morphologies* and *Kinodynamics*, but they can be deduced from previous discussions.

The diagram focuses on structural relationships between *Knowledge Entities* that are according to previous definitions.

- A *Body* evolves in a set of *Environments*, it can support a set of *Infrastructure* elements and it participates to a set of *Interactions*. A *Body* can be decomposed into smaller *Bodies* as well as *Infrastructure*.
- An *Environment* can be decomposed into a set of more spatially restricted *Environments*.
- A *Phenomenom* occurs within an *Environment* and is detected thanks to a set of *Interactions*.
- An *Interaction* occurs within *Environments*.
- A *Behavior* is activable according to a set of possible *Interactions* and can be observed when the robot realizes a given set of *Actions*.
- An *Action* commands a *Body* and has, as goals, a set of *Objectives*.
- An *Objective* references the target *Environement* to express its spatial localization.

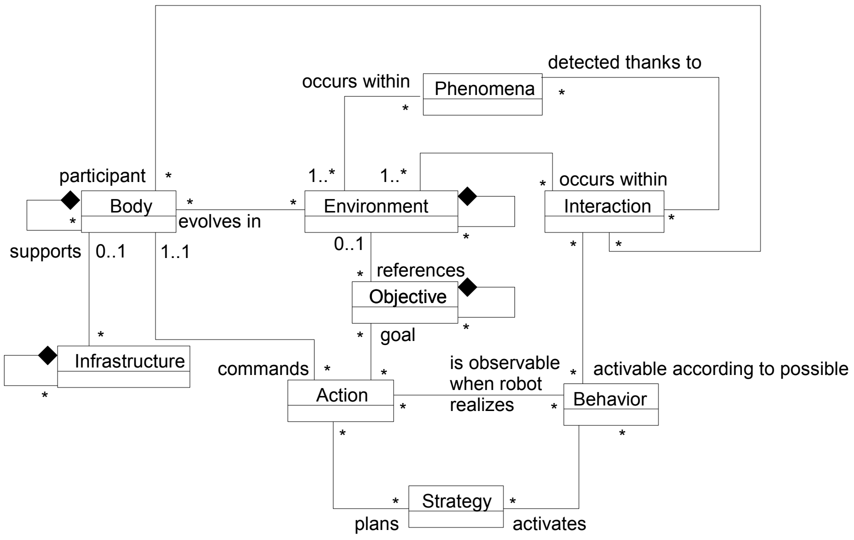


Figure 5. Meta-model : diagram detailing important *Knowlegde Entities*

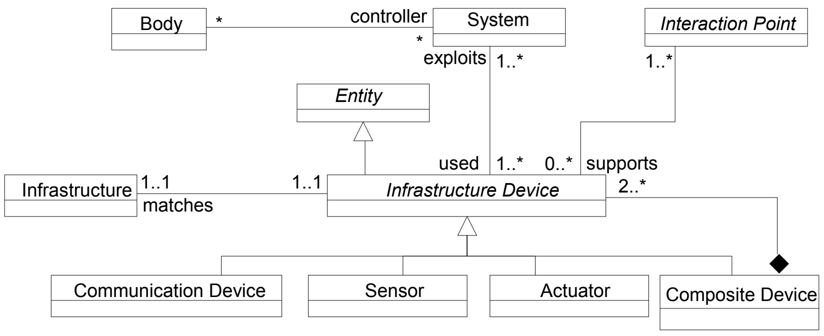


Figure 6. Meta-model : diagram defining entities used in detailed architecture description

- A *Strategy* plans a set of *Actions* to realize, and according to this set it activates a set of *Behaviors*.

This relations in the meta-model will be traduced by links between *Knowledge Entities* instances in a control architecture model. These links will be stereotyped with `<< ... >>` label. For instance for links between *Body* and *Environment*, the stereotype `<<evolvesin>>` will be used. Composition links between *Knowledge Entities* in the meta-model are traduced by composition links in a model.

3.2.3. Complete Control Architecture Description

Since a precise description of control architectures, like it is presented in figure 3, has to be possible, the relation between *Decisional Entities* and *Physical Infrastructure Elements* have to be described. This is the purpose of the diagram of figure 6. This part of the meta-model more specifically details *Systems* according to their relation with operative part they control and physical elements they use to control it.

The relation between a *System* and the robot operative part it controls it expressed according to the *controller* relation between *Body* and *System* meta-classes. A constraint, that is not expressed in the diagram, is that all *Decisional Entities* it contains participate to the control of the same *Body* or sub-parts of it. To describe the way a *System* controls a *Body*, the diagram introduces the abstract class *Infrastructure Device* that is itself specialized into four concrete classes :

- *Communication Device* (square with *C* label cf. fig. 3) that represents a device that the robot uses to communicate with human operators or other robot, for instance a Wifi device.
- *Sensor* (square with *S* label cf. fig. 3) that represents a physical sensor, for instance a joint position sensor.
- *Actuator* (square with *A* label cf. fig. 3) that represents a physical actuator, for instance a joint position command actuator.

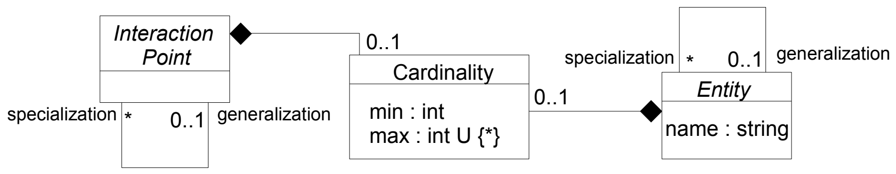


Figure 7. Meta-model : diagram defining Genericity, Refinement and Optionality management

- *Composite Device* (represented with a white rectangle around other devices in figure 3) that represents a set of simpler devices that can be grouped according to a given infrastructure decomposition. For instance, it can represent all joint sensors and all joint actuators of a same robot arm or all devices of a robot.

All *Infrastructure Devices* support a set of *Interaction Points* (restricted to *Perceptions* for *Sensors* and *Reactions* for *Actuators* by which they can interact with *Decisional Entities*. A *Composite Device* just supports the set of *Interaction Points* of its contained *Infrastructure Devices*. A *System* so exploits a given set of *Infrastructure Devices* by putting in relation its *Interaction Points* with theirs. For example, a *Manipulator System* exploits (at least) the *Composite Device* that represent the sensors and actuators of arm. It has to be noticed that an *Infrastructure Device* can be used by many *Systems* (of course it has to be done with care). Finally, diagram shows that an *Infrastructure* knowledge entity refers to a corresponding *Infrastructure Device*.

3.2.4. Genericity, Refinement and Variability

Now that all primitive structures of the language used to describe control architectures have been defined, this subsection states the management of control architecture solutions description. To describe architectural solutions, mechanisms allowing to deal with different degrees of genericity-specialization are necessary Mechanisms introduced in the language are defined in the diagram of figure 7. These mechanisms are all defined thanks the *Cardinality* class, inspired from UML [14] cardinalities.

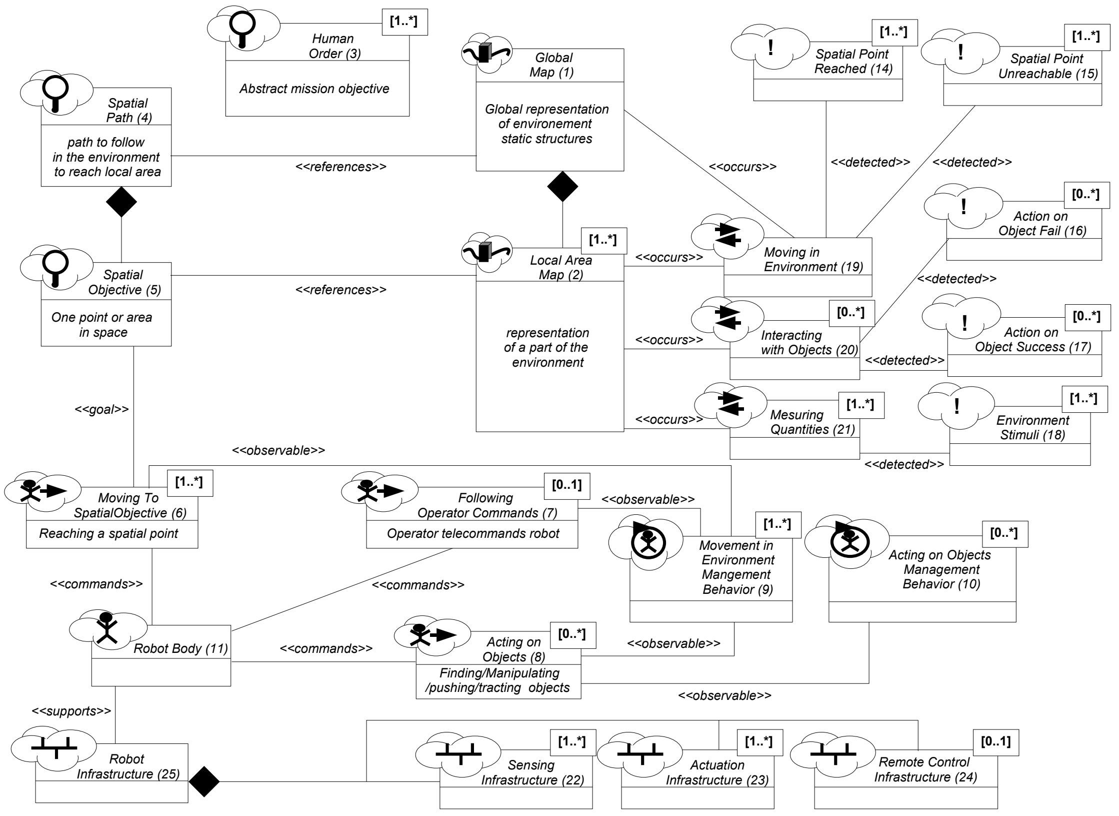
Individually, a *Cardinality* just represents minimal and maximal amounts, where maximal amount can be infinite number (using * value). When a *Cardinality* is associated to an *Entity*, this means that this *Entity* can be refined, in an architecture that conforms to the solution, as many times as allowed by the *Cardinality* maximal and minimal value. So, *Cardinality* is used to deal with the genericity of control architecture solutions. For example, when considering a team of robots where each robot has the same control architecture, the architectural solution is expressed with a single *System* and its associated *Cardinality* that expresses the minimal and maximal number of team members. If no *Cardinality* is explicitly associated to an *Entity*, this means that this latter has a `[1..1]` cardinality. When a cardinality is associated to an *Entity* having a composite relation with other *Entities* (e.g. *System*, *Body*, etc.), this *Cardinality* is applied to all its internal description, allowing so to express a possible duplication of the *Entity* internal structure. For example, when a cardinality is associated to a *System*, this means that each of its contained *Activity* and *Coordination* can be duplicated as many times as specified by the *System*'s *Cardinality*. *Cardinalities* can also be associated with *Interaction Points*, meaning these points can be duplicated and refined when their source and or target *Entity* are refined.

Variability in a model allows the user to express different possible choices in its design. In feature models [15] used to express product lines, variability is expressed in two ways : optional features and group features. An optional feature corresponds to a feature that can or not be present in the resulting product. A group feature corresponds to a possible choice between a set of feature. In the present work, optionality means that an *Entity* is present or not in a refined control architecture. It is expressed with a `[0..1]` cardinality applied on this *Entity* or *Interaction Point* and can be extended to `[0..n]` cardinalities, whatever `n` value is. In a first time, the language only incorporate description feature for optionality and does not deal with group cardinalities. When an optional entity disappears in a refinement all its interaction points (for decisional entities) or relations (for knowledge entities) disappear. Furthermore, when a *Coordination* has only one possible participating entity in a refinement, it disappears.

Refinement is expressed in the meta-model with the *generalization-specialization* relationship on *Entity* and *Interaction Point* classes (cf. fig. 7), that is similar to the UML class *specialization* feature. This relation allow to refine a set of *Entities* to adapt an architural solution to a control architecture of a given robot. So the refinement is restricted according to cardinality. When a *Knowledge Entity* is refined, its relations with other *Knowledge entities* are themselves specialized with relations between related specialized *Knowledge Entities*. By default, in an architectural solution, all relations supports a `[0..*]` cardinality, meaning they are optional but can be duplicated many times.

4. Example

This section presents the use of the modeling language for the description of Aura Architectural solution. This example is defined according to a personal understanding and interpretation of Aura solution, based on its related bibliography. This is an important precision: since only the authors deeply know Aura, their initial viewpoints could be unintentionally



Human Order (3)
[1..*]
Abstract mission objective
Global Map (1)
Global representation of environement static structures
Spatial Path (4)
path to follow in the environment to reach local area
<<references>>
Spatial Point Reached (14)
[1..*]
Spatial Point Unreachable (15)
[1..*]
<<occurs>>
<<detected>>
<<detected>>
Action on Object Fail (16)
[0..*]
Spatial Objective (5)
One point or area in space
<<references>>
Local Area Map (2)
[1..*]
representation of a part of the environment
<<occurs>>
Moving in Environment (19)
<<detected>>
Interacting with Objects (20)
[0..*]
<<occurs>>
Action on Object Success (17)
[0..*]
<<detected>>
<<occurs>>
Mesuring Quantities (21)
[1..*]
Environment Stimuli (18)
[1..*]
<<detected>>
<<goal>>
<<observable>>
Moving To SpatialObjective (6)
[1..*]
Reaching a spatial point
Following Operator Commands (7)
[0..1]
Operator telecommands robot
<<observable>>
Movement in Environment Mangement Behavior (9)
[1..*]
Acting on Objects Management Behavior (10)
[0..*]
<<commands>>
<<commands>>
Robot Body (11)
<<commands>>
Acting on Objects (8)
[0..*]
Finding/Manipulating /pushing/tracting objects
<<observable>>
<<observable>>
<<supports>>
Robot Infrastructure (25)
Sensing Infrastructure (22)
[1..*]
Actuation Infrastructure (23)
[1..*]
Remote Control Infrastructure (24)
[0..1]

Figure 8. Aura Architectural Solution : Knowledge dimension (a)

not respected. So these examples should be seen as an illustration of the use of the modeling language rather than a "definitive" opinion on the way Aura architectures are designed.

Aura is a generic and abstract control architecture solution which merges a reactive approach for low-level control design with a hierarchical "deliberative" approach for high-level control design. It has been chosen because Aura is a very complex and general architectural solution which integrates preoccupations like teleoperation and learning.

The example mainly refers to [4]. The architectural solution is decomposed in two dimensions : the knowledge dimension presented in figures 8 and 9 and the decisional dimension presented in figure 10. The relations between these two dimensions are expressed thanks to numbers in parenthesis associated to knowledge entities (cf. fig. 8 and 9) and used in decisional entities (cf. 10).

4.1. Knowledge Dimension

The knowledge dimension decomposes (cf. fig. 8) the generic knowledge entities used in the Aura architectural solution. *Human orders* are abstract mission objectives given by human operators. The *Human Order* knowledge can be refined many times to describes different types of mission objectives. The *Global Map* is a global representation of static elements of the environment. It is decomposed into *Local Areas*, representing for instance things like rooms or corridors. The *Local Area* knowledge can be refined many times to describes these different types of *Local Area* if it is of importance. The *Spatial Path* is an objective that defines the path the robot has to follow in the environment. It is decomposed into a set of *Spatial Objectives* representing important places where the robot has to accomplish specific objectives. The *Spatial Objective* entity can itself be refined into specific objectives for specific *Local Areas*. Three generic knowledge entities are used to represent interactions: *Moving In Environment*, *Interacting with Objects* and *Measuring Quantities*. All these *interactions* occurs in *Local Area*, except *Moving In Environment* that also occurs within *Global Map* (i.e. going from one local area to another). A set of *phenomena* are detected thanks to these interactions: *Spatial Point Reached* and *Spatial Point Unreachable* detected thanks to *Moving In Environment*, *Action on Object Fail* and *Action on Object Success* thanks to *Interacting with Objects*, *Environment Stimuli* thanks to *Measuring Quantities*. Figure 8 shows that *Interacting with Objects* interaction and related phenomena are optional, in the sense that Aura Robot does not necessarily have means to manipulate objects nor have for mission to transport object in the environment. Other Interactions and phenomena are not considered to be optional (they have to be refined at least one time in a control architecture).

Figure 8 defines three types of actions: *Moving to Spatial Objective* which consists for the robot to go from current point to a spatial objective, *Following Operator Commands* which consists for the robot to be teleoperated, *Acting on Objects* which consists to make an action on an object to obtain a given result (e.g. catching an object, throwing it in

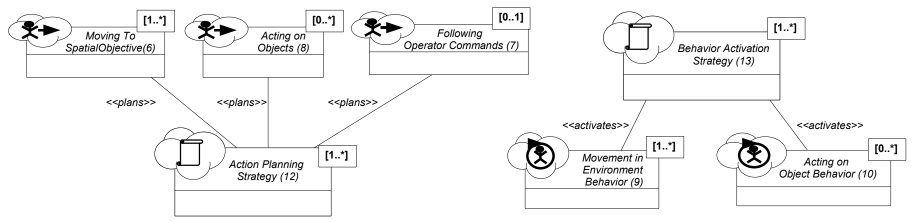

[1..*]
Moving To SpatialObjective(6)
[0..*]
Acting on Objects (8)
[0..1]
Following Operator Commands (7)
<<plans>>
<<plans>>
<<plans>>
Action Planning Strategy (12)
[1..*]
[1..*]
Behavior Activation Strategy (13)
<<activates>>
<<activates>>
[1..*]
Movement in Environment Behavior (9)
[0..*]
Acting on Object Behavior (10)

Figure 9. Aura Architectural Solution : Knowledge dimension (b)

a bin, etc.). All actions are considered to be optional except *Moving to Spatial Objective*, because Aura is initially a solution for robot mobility. Two knowledge entities generalize all behaviors of robot: *Movement in Environment Behavior* that generalizes behaviors like "approaching spatial point" or "obstacle avoidance" and *Acting on Objects Behavior* like "catching nearest waste". For reason of conciseness of the model, relations between interactions and behaviors are not represented. For example, *Acting on Objects Behavior* should be linked with a `<<possible>>` relation with all three interactions. All actions commands a unique *Robot Body* because Aura does not propose a decomposition of the knowledge of robot operative part into smaller controlled parts. The *Robot Body* supports a *Robot Infrastructure* itself composed of sensing and acting infrastructural elements and optionally a remote control infrastructural element. Control strategies are defined in figure 9. As the decomposition of knowledge of Aura architectural solution has been understood there are two categories of strategies represented by two generic knowledge entities: *Action Planning Strategy* defines an action planner and *Behavior Activation Strategy* defines a way a set of behaviors are activated and merged.

Next subsection shows the way all these entities are used in the decisional process decomposition of Aura architectures.

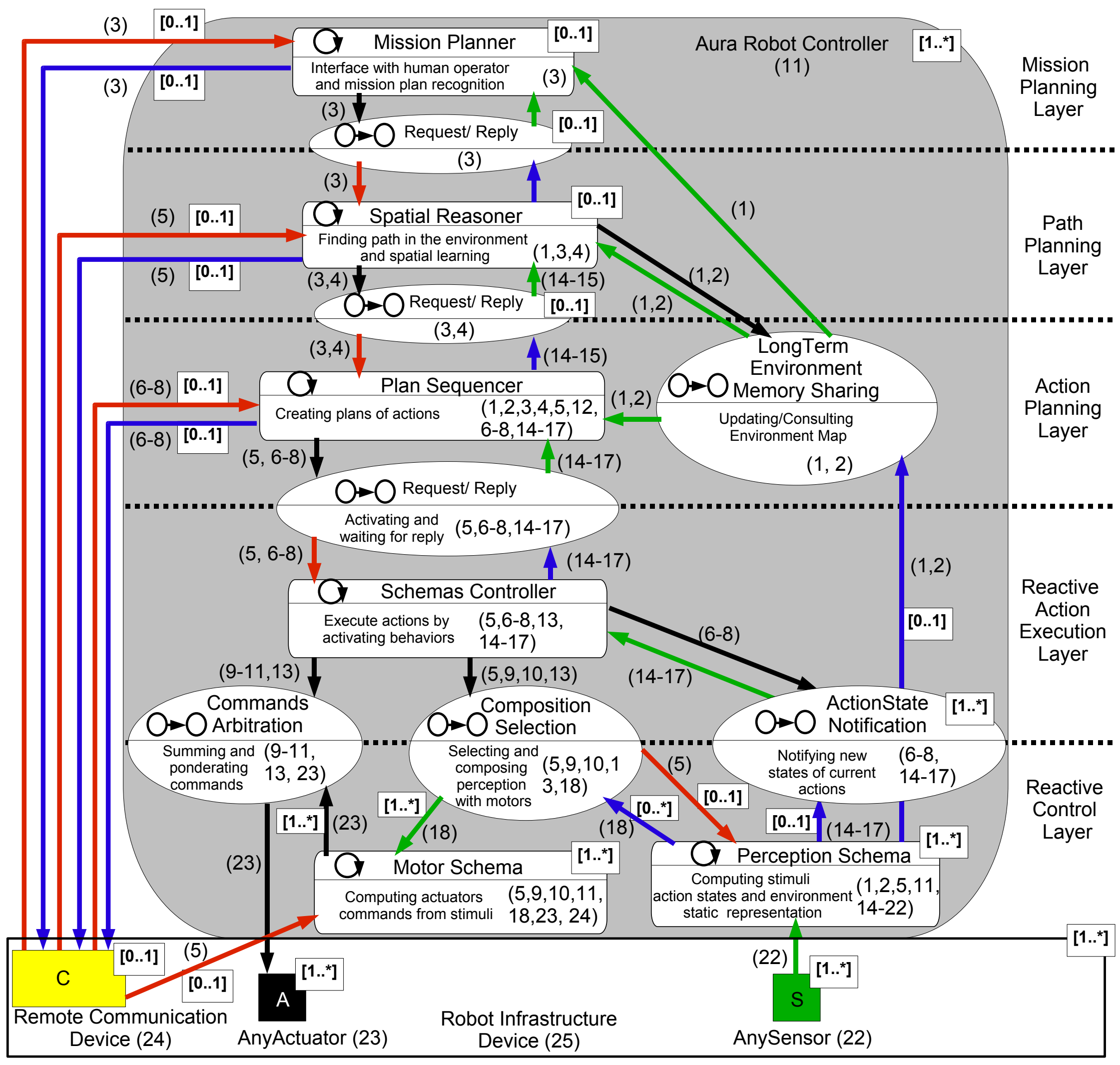
4.2. Decisional Dimension

The decomposition of robot architectures into subsystems does not exist in Aura: each robot is associated to a single system. System decomposition arises as soon as a group of collaborative robots is considered, each system (i.e. robot) being attached on its own infrastructure (cf. fig. 10). Robot system is decomposed into five layers, two reactive layers and three deliberative layers.

At the top layer there is the *Mission Planner* activity, in charge of collecting user intentions (i.e. mission long term goals and constraints) represented with *Human Order* (3) knowledge entity. At the layer below, the *Spatial Reasoner* activity receives requests from the *Mission Planner* to define the *Spatial Path* (4) (sequence of *Spatial Objectives*(5)) the robot must or can follow to achieve *Human Order* (3). Once a path is defined the *Plan Sequencer* activity is invoked to define the sequence of actions (6-8) required to follow the path and to achieve *Human Order* (3). For example, if the *Human Order* is to clean a building, the first action sequence would be: "go to room 1" (6), "collect waste"(8) and "put waste in the bin" (8) if the *Spatial Path* path is "room1, room2, room3, etc.". The action sequence corresponds to a state diagram where states are actions to perform and transitions are action changes. Transitions are associated to specific phenomena (14-17) that enable the state change. The planning itself is defined according to a given *Action Planning Strategy* (12) and according to *Global Map* and *Local Area* knowledge.

The activity entities of the deliberative layers interact around a *Long Term Environment Memory Sharing* coordination entity to consult and update *Global Map* and *Local Areas* knowledge (1,2).

Once a sequence of actions has been defined, the *Plan Sequencer* invokes the *Schema Controller* activity of the *Reactive Action Execution Layer* to realize each action (6-8). To this end the *Schema Controller* defines a *Behavior Activation Strategy* for each action to realize. *Schema Controller* interacts with *Perception Schemas* and *Motor Schemas* activities of the *Reactive Control Layer*. *Perception Schemas* activities are responsible of the production of *Stimuli* (18) or other action execution related phenomena (14-17) from sensors data. *Stimuli* are phenomena that contain partial instantaneous representations of the *environment* (1,2) or *robot body* (11). Other phenomena (14-17) are representing interesting state of actions execution. *Perception Schemas* can also produce long term Environment (1,2) representations (e.g. map of a room, update of the *Global Map*). *Motor Schema* activities put in place specific behaviors (9,10). Each *Motor Schema* activity is viewed as a control low, computed using *stimuli* (18) and *Robot Body* (11), to obtain a given behavior (9,10), as for example "obstacle avoidance". *Schema Controller* activity coordinates *Perception Schemas* and *Motor Schemas* in different ways. First, it translates the action execution request into a composition of Schemas (cf. *Composition Selection*): *Motor Schemas* are activated according to the behavior they represent ; *Stimuli* generated by *Perception Schemas* are redirected to *Motors Schemas* following a predefined *Behavior Activation Strategy* (13) and *Perception Schemas* can be configured with a *Spatial objective* (5). Second, some *Perception Schemas* are activated to generate action execution status (cf. *Action State Notification*). *Schema Controller* uses these phenomena to know if the current action has been (or cannot be) realized. It can then reply to the *Plan Sequencer* to indicate if the action succeeded or failed; if the action failed it tries to re-plan a sequence of actions or it indicates the *Spatial Reasoner* that the path cannot be followed. *Perception Schemas* can also interact with the higher-level activities by updating the *Long Term Environment Memory*. Finally, the *Schema Controller* sums and balances the commands to motors generated by activated



Mission Planner
Interface with human operator and mission plan recognition
Aura Robot Controller (11)
Mission Planning Layer
Request/ Reply
Spatial Reasoner
Finding path in the environment and spatial learning
Path Planning Layer
Plan Sequencer
Creating plans of actions
LongTerm Environment Memory Sharing
Updating/Consulting Environment Map
Action Planning Layer
Activating and waiting for reply
Schemas Controller
Execute actions by activating behaviors
Reactive Action Execution Layer
Commands Arbitration
Summing and ponderating commands
Composition Selection
Selecting and composing perception with motors
ActionState Notification
Notifying new states of current actions
Reactive Control Layer
Motor Schema
Computing actuators commands from stimuli
Perception Schema
Computing stimuli action states and environment static representation
Remote Communication Device (24)
AnyActuator (23)
Robot Infrastructure Device (25)
AnySensor (22)

Figure 10. Aura Architectural Solution : Decision dimension

Motor Schemas (cf. *Command Arbitration*) according to the respective importance of behaviors in the chosen *Behavior Activation Strategy*. Once done, the command vector is applied to robot's motors. Human machine interaction can take place in many ways in an Aura architecture. Human can send *Human Order* to the *Mission Planner* at the top most level, it can directly send *Spatial Objectives* to the *Spatial Reasoner*, it can send actions to execute to the *Plan Sequencer* and finally it can directly interact with a specific *Motor Schema* to control robot movements.

5. Conclusion

This paper has argued on the necessity to define a dedicated language for communicating and understanding architectural choices in robotics. It proposed such a modeling language that embeds conceptual and terminological properties of the domain. It allows expressing architectural specificities by providing adequate abstractions and by promoting the importance of control organization thanks to different abstract types of entities: System, Activity, Coordination and Knowledge. An example of use of this model have been developed on a well-known architecture.

Future work aims for the design of decision mechanisms of Activity entities and that of protocols of Coordination entities. Simple concepts and terminologies have to be defined to easily express their respective properties. In the same way, more detail should be integrated to the different Knowledge entities types, to describe more precisely their respective models.

Another important point is a better understanding of genericity and refinement mechanisms. The refinement mechanism and cardinalities' impact on refinement should be formalized to avoid ambiguities. It would also be usefull to provide a way to describe alternative possibilities. In that sense, feature models [15] is a good source of inspiration because it provides structures, like group features and include/excluse constraints, to deal with it.

The short term goal of this work is to propose a precise frame to compare existing control architecture solutions and to highlight their advantages and limitations. In a longer term, the goal is to find commonalities and variations between all the main designs proposed in the domain.

References

[1] R. Alami, R. Chatila, S. Fleury, M. Ghallab, and F. Ingrand. An architecture for autonomy. *International Journal of Robotics Research, vol.17, n°4 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] R.C. Arkin and T. Balch. Aura : principles and practice in review. Technical report, College of Computing, Georgia Institute of Technology, 1997.

[5] 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.

[6] 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.

[7] 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.

[8] D. Brugali and P. Salvaneschi. Stable aspects in robot software development. *Advanced Robotic Systems, vol. 3, n° 1*, pages 17–22, 2006.

[9] 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.

[10] B. Espiau. La peur des robots. *Science Revue , Hors série n°10, p.49*, mars 2003.

[11] L. Fluckiger and H. Utz. Lessons from applying modern software methods and technologies to robotics. In *Software Integration and Development in Robotics, SDIR07 at ICRA07 (International Conference of Robotic and Automation)*, 2007.

[12] E. Gama, R. Helm, R. Jonhson, J. Vlissides, and G. Booch. *Design Patterns : Element of Reusable Object-Oriented Software*. Addison-Wesley Professional Computing, 1995.

[13] E. Gat. On three-layer architectures. *A.I. and mobile robots, D. Korten Kamp & al. Eds. AAAI Press*, 1998.

[14] Object Management Group. Uml ressource page, http://www.uml.org/.

[15] K. Kang, S. Cohen, J. Hess, W. Nowak, and S. Peterson. Feature-oriented domain analysis (foda) feasability study. Technical Report CMU/SEI-90-TR-21, Carnegie Mellon University, Pittsburgh, PA, November 1990.

[16] J. Maaß, N. Kohn, and J. Hesselbach. Open modular robot control architecture for assembly unsing the task frame formalism. *Advanced Robotic Systems, vol. 3, n° 1*, pages 1–10, 2006.

[17] 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.

[18] M. Mataric. Situated robotics, in encyclopedia of cognitive science, 2002.

[19] M. Mataric. Behavior-based control: example from navigation, learning and group behavior. *Journal of Experimental and Theoritical Artificial Intelligence 9*, pages 323–336, 1998.

[20] M. Mernik, J. Heering, and A.M. Sloane. When and how to develop domain specific languages. *ACM Computing Surveys, Vol.37, n°4*, pages 316–344, 2005.

[21] 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.

[22] R. Passama, D. Andreu, C. Dony, and T. Libourel. Overview of a new robot controller development methodology. In *1st conference on Control Architecture of robots (CAR'06), pp. 145-163*, 2006.

[23] J.K. Rosenblatt. Damn : a distributed architecture for mobile navigation. Technical report, The Robotic Institute, Canergie Mellon University, 1997.

[24] C. Schlegel. A component approach for robotics software: Communication patterns in the orocos context. *18 Fachtagung Autonome Mobile Systeme (AMS)*, pages 253–263, 2003.

[25] 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.

[26] 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, n°2, pp.249-277*, June 1996.

[27] D.B. Stewart. Designing software components for real-time applications. In *2001 Embeded System Conference*, San Francisco, 2001.

[28] C. Urmson, R. Simmons, and I. Nesnas. A generic framework for robotic navigation. In *IEEE Aerospace Conference, vol. 5, pp. 2463-2470*, 2003.

[29] 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.