FACULTAD DE INFORMÁTICA

UNIVERSIDAD POLITÉCNICA DE MADRID

TESIS DE MÁSTER

MÁSTER DE INVESTIGACIÓN EN INTELIGENCIA ARTIFICIAL

IMPROVING ROBUSTNESS IN ROBOTIC NAVIGATION BY USING A SELF-RECONFIGURABLE CONTROL SYSTEM

AUTOR: Arturo Bajuelos Castillo

TUTORES: Nik Swoboda y Ricardo Sanz Bravo

SEPTIEMBRE, 2011

MEJORANDO LA ROBUSTEZ EN NAVEGACIÓN ROBÓTICA USANDO UN SISTEMA DE CONTROL AUTO-RECONFIGURABLE

Arturo Bajuelos Castillo

4 de septiembre de 2011

Acknowledgements

I would start by thanking my tutors, Nik and Ricardo, which gave me the opportunity and the means that made possible all the work related with this thesis.

I would not had the energy nor the motivation to accomplish this work without the constant support and love of my parents, my sister and my girlfriend.

I would also like to thank all my friends for always being there and for knowing how to cheer me up and make me happy.

To my laboratory colleagues, which were always willing to teach me when I needed to learn.

A would also like to express a special gratitude to my boss and my work colleagues for believing in my work and understanding my situation.

My biggest thank would have to go to Carlos. Without his constant support, help, guidance and patience, I would not be able to complete this thesis in any of its aspects. It was an honor to have him as work colleague and as a good friend.

Resumen

En los tiempos modernos estamos rodeados de una gran diversidad de sistemas tecnológicos que paulatinamente se han integrado en nuestra sociedad. Sin embargo, muchos de estos sistemas no son capaces aún de hacer frente a entornos complejos, dinámicos e inciertos sin la intervención humana ni alcanzar los niveles requeridos de fiabilidad y robustez. Un buen ejemplo de estos sistemas son los de la industria de la robótica. La mayoría de los robots que realmente hacen un trabajo útil para nosotros consiste en robots industriales. Estos robots realizan operaciones en entornos rigurosamente controlados y con misiones muy bien definidas. Fuera de los laboratorios de investigación resulta prácticamente imposible encontrar otros robots con cierto grado de autonomía. Este trabajo de investigación hace frente a este problema, intentando aumentar la adaptabilidad de estos sistemas utilizando un enfoque general.

En este trabajo se describe el sistema de control que ha sido desarrollado para un robot móvil autónomo que realiza una misión de vigilancia y donde se considera que pueden ocurrir fallos en alguno de sus dispositivos. Se define la misión del robot y la arquitectura de control diseñada que es la contribución principal de este trabajo de investigación. La arquitectura propuesta está inspirada en la arquitectura cognitiva *The Operative Mind* e incluye una capa de meta-control cuya función es controlar y reconfigurar el sistema caso sea necesario. Este enfoque puede parecer muy similar al que se aplica a sistemas de tolerancia a fallos; sin embargo, la capa de meta-control se define aquí con generalidad absoluta, y puede ser fácilmente ampliada para mejorar aún más la solidez y la autonomía del sistema de control, mediante la detección de eventos del sistema que no necesariamente están relacionados con fallos. Este sistema de control se implementó y los experimentos se realizaron con el objetivo de verificar que el robot es capaz de cumplir su misión gracias a sus capacidades de auto-reconfiguración. Otros experimentos revelaron que esta capa de meta-control puede ser fácilmente mejorada, aumentando la eficiencia del sistema.

Abstract

In these modern times, we are surrounded by many technical systems that are successfully integrated in our society. However, they are still not able to deal with complex, dynamic and uncertain environments without human intervention. Most of these control systems have not reached yet the required levels of reliability and robustness. A good example is represented by the robotic industry. The majority of robots that actually do some useful work for us consist of industrial robots that perform operations in strictly controlled environments and very well defined missions. However, outside the labs we have not seen many other robots doing anything autonomously but wander. This research work concerns with this topic: augmenting systems resilience using a general approach.

In this work, a robot control system is developed for an autonomous mobile robot that executes a simple but challenging indoor surveillance mission in which device failures may occur. After the full definition of the mission, the control architecture is designed, which consists in the main contribution of this research work. This architecture was inspired by the general cognitive architecture *The Operative Mind*, and includes a meta-control layer. The meta-control layer is intended to monitor and reconfigure the system if necessary. This approach may appear very similar to the ones applied to fault-tolerant systems. However, the meta-control layer is defined here with absolute generality, and can be easily augmented to further improve the robustness and the autonomy of the control system, by detecting system's events that are not necessarily related with faults. The control system was implemented and experiments were done to verify that the robot can accomplish its mission completely thanks to the self-reconfiguration capabilities of the system brought by the meta-control layer. Other experiments revealed that this meta-control layer can be easily upgraded to enhance the efficiency of the system.

Table of Contents

Table of Figures

1 Introduction

1.1 Background and Motivation

In a time when we search for fully autonomous systems and machines, the biologic mind – human or animal – is still the only reference of an autonomous control system robust to dynamic and uncertain environments.

The last few decades have witnessed huge developments and integrations of technical systems into our society. Most of these systems replaced human labor, while some other extended the technological frontiers enabling us to achieve things that were unreachable or unimaginable before. In modern societies, we face daily many of these systems, despite not being aware of them. Airplanes, cars or chemical plants, along with electricity networks, telecommunications and other supporting facilities are some examples of these. However, many of the new demands on technical systems are extremely difficult to achieve if one wants to maintain or increase other non-functional requirements such as reliance, robustness or even autonomy. A good example of this fact is the robot industry. Some years ago, many predicted that today we would have fully autonomous robots cooperating with us as our artificial intelligent partners. Nonetheless, currently most of the robots that are successfully integrated into our societies exist as industrial robots, which basically correspond to reprogrammable manipulators with specific tasks that work in production plants. Typical applications of these robots include welding, painting, assembly, along with pick and place. Despite being extremely efficient in their tasks, there are still many responsibilities in industry only trusted to humans. Outside factory plants, robots doing anything autonomously more complex than wandering –i.e. Roomba[1] – have still not left lab environments. They were foreseen to guide people in museums, patrol security areas, drive us to our destinations as autonomous vehicles, or perform exploration in Mars. And to some degree they already do. But their dependence upon human supervision makes them less efficient. These robots can be seen as examples of demanded technical systems that have not yet achieve the desired levels of autonomy, robustness and reliance.

Fault-tolerant systems have been developed to overcome the failures that may occur in technical systems, either because of the interaction with the environment or originating from the system itself. They are usually implemented as technical systems whose reliability is essential. However, as these types of systems are specified nowadays, one has to capture, into a model, an almost full knowledge of the environment and the system itself. An example of this can be appreciated in the fault-tolerant architecture described in (Blanke, et al. 2006). Therefore, as with other *conventional* control systems, an accurate formal model of the system must be specified in order to implement them successfully.

In the past decades, several Artificial Intelligence and Soft Computing techniques were used when a control system's model could not be specified and classical techniques

[1] http://store.irobot.com/home/index.jsp

(such as PID[2] or state space control) could not be applied. With some collected data, the machine could automatically find an approximate model. However, for a system operating in partially (or totally) unknown and/or unpredictable environments that handles high levels of uncertainty, it is not enough to individually apply these techniques. One must organize them into a previously designed structure, which is represented by the system's control architecture. This thesis is based on the premise that the definition of a proper control architecture is required to successfully implement a complex system which interacts with an unknown and uncertain environment.

Several architectures for control systems have been proposed when high demand on technical systems started to appear. In particular, since the 1970's, many agent architectures were designed and implemented in several control system for autonomous agents. These architectures can be mainly divided into three categories: *deliberative*, *reactive*, and *hybrid* architectures (Wooldridge and Jennings 1995). However, it is important to analyze agent architectures from a cognitive point of view because the biological mind is a strong referent for a successful autonomous robust control system. Cognitive architectures have emerged as architectures that propose computational processes that act like certain cognitive systems (most often, like a person). They bring us closer to the general problem of developing a truly intelligent and autonomous agent. By being inspired by biological cognitive phenomena, or by trying to address the general problem of intelligence, they have been considered a respected approach with some real implementations in physical agents: see (Albus, et al. 2002). The presented work is highly related to cognitive architectures, with the aim to develop control architectures that lead to the implementation of more robust and reliable systems, capable to interact with dynamic and unpredictable environments.

1.2 Contributions

This thesis contains a description of the development of a control system, which was successfully implemented in an autonomous mobile robot. However, the main contribution of this work is the designed architecture, used to implement the control system. This architecture is based on the general cognitive architecture *The Operative Mind* (Hernandez, Lopez and Sanz 2009) integrated in the ASys Project[3] that include a meta-control layer whose function is to monitor and reconfigure the system if necessary. Unlike fault-tolerant architectures, the meta-control layer is defined here with absolute generality, and can be easily augmented to further improve the robustness and the autonomy of the control system by detecting system's events that are not necessarily related with faults.

Thus, the general objective of this work is to specify a guideline of how to increase the reliability of an autonomous system by inserting self-introspection abilities. This is done by designing a complete architecture that defines a control system capable of self-

[2] PID is an acronym for "*Proportional-Integral-Differential*", and consists in the most used feedback control design. For more information, see (Araki 2000).

[3] The ASys project is long term project of the Autonomous Systems Laboratory (ASLab) research group of the Technical University of Madrid.

monitoring and self-reconfiguration. The system is implemented in an autonomous mobile robot and a mission which requires self-reconfiguration is specified. Experiments results show the necessity of self-awareness and self-reconfiguration for the complete execution of the mission.

To achieve the research objectives, the following tasks have been performed:

- Study of agent architectures and cognitive architectures, as well as the necessity of self-introspection in today's control systems;

- Specification of a proper mission for the control system, having in mind the hardware platform available and the work requirements. This mission must require system's self-reconfiguration;

- Design of a full architecture for the control system that is suited for the execution of the mission. This architecture must specify components that are capable of self-monitoring and self-reconfiguration;

- Implementation of the designed architecture using the available hardware and software platform;

- Realization of several experiments and extraction of meaningful results aimed at showing that self-introspection abilities are essential for the execution of the mission thus allowing to increase the system's efficiency;

1.3 Structure of this document

Chapter 2 presents an overview of the state of the art of the scientific areas related with this work, such as Agent Architectures, Cognitive Architectures, Fault Tolerant Systems and some of the most popular techniques used in Mobile Robotics. It also describes some popular software robotic platforms, which are alternatives to the one used for this work. This chapter is important to understand the following chapters, since it captures essential knowledge of the context of this work. In Chapter 3, the general architecture *The Operative Mind* is described, as well as its surrounding project – ASys. This chapter presents some of the theoretical framework in which the developed architecture is based. In Chapter 4, the available hardware platform used to implement the developed system is described, as well as an overview of the Robotics Operating System (*ROS),* the robotic software platform used. Chapter 5 presents a detailed description of the chosen mission that the robot has to accomplish. In Chapter 6, the developed architecture for the control system is presented. This chapter is the core of this thesis, since the major contribution of this work relies on the description of the developed architecture. Chapter 7 presents a description of the implemented robot control system. In the Chapter 8, the description of the experiments made, the measures taken and their analysis can be found. Finally, Chapter 9 contains the conclusions of the work, with special emphasis in the results obtained in the experiments and a reference to some future works.

2 State of the Art

The following subsections try to capture all the necessary knowledge to contextualize this work in its associated domains. An overview of the state of the art of the areas related with this work is presented, such as Agent Architectures, Cognitive Architectures (as a subset of Agent Architectures) and Fault-Tolerant Systems. The subsections 2.4 and 2.5 serve as a presentation and review of the main techniques of autonomous mobile robots (some of them used in this work) and some popular robotic software platforms that consists of alternatives to the software platform used for this work (which will be detailed in 4.2 Software Platform – ROS).

2.1 Agent Architectures

Agent architectures are studied in several scientific areas, namely in Autonomous Robotics. Maes considers an agent architecture to be:

"[A] particular methodology for building agents. It specifies how the agent can be decomposed into the construction of a set of component modules and how these modules should be made to interact. The total set of modules and their interactions has to provide an answer to the question of how the sensor data and the current internal state of the agent determine the actions ... and future internal state of the agent." (Maes 1991)

For autonomous robots, a control architecture is a blueprint for intelligent control systems that provide some intelligence, robustness or autonomy to a robot. They can be metaphorically compared to the "brain" behind an autonomous robot. In a less abstract view, they can be seen as the element that provides to a robot the capability to extract information from its environment and use knowledge about its world to move safely in a meaningful and purposeful manner. They can also refer to the way in which the sensing and the action of a robot are coordinated.

Nowadays, we can choose our robot control architecture from a wide defined spectrum of control. It is important to know that there is no such thing as a perfect architecture for all types of robots or for every kind of environment. The control architecture adequacy to a robot depends heavily on the mission's characteristics, environment and the robot software and hardware resources. Control architectures are grouped in three defined approaches, each with different characteristics and control trade-offs:

- **Deliberative Architectures** – adequate when long-term planning and reasoning is essential.

- **Reactive Architectures** – adequate for agents situated in highly dynamic environments where quick answers are essential.

- **Hybrid Architectures** – adequate for agents situated in unknown and changing environments or when a reactive behavior is as needed as long-term planning.

In the following section, a description of each of the above architectures is presented, as well as some of the most important landmarks and related works.

2.1.1 Deliberative Architectures

Deliberative Architectures (also considered as the classical or symbolic approach) appeared in the early 1970's and consisted in the first approach for the design of agents within AI. According to Wooldridge, a deliberative agent is *"one that possesses an explicitly represented, symbolic model of the world, and in which decisions (for example about what actions to perform) are made via symbolic reasoning"* (M. Wooldridge 1995). Generally, its purest expression proposes that agents use explicit logical reasoning in order to decide their actions, while possessing an internal image of the external environment. This foundation of the symbolic AI paradigm is supported by the *physical-symbol system hypothesis* (Newell and Simon 1976). The term "deliberative agent" derives from Genesereth's use of the term *"deliberate agent"* to mean the specific type of this symbolic architecture (Genesereth and Nilsson 1987).

The main idea of a deliberative agent is to use logic to encode a theory stating the best action to perform in any given situation. This idea of an agent capable of logical reasoning is "*highly seductive"*, as stated by (Wooldridge and Jennings 1995), because to get an agent to realize some theory one might naively suppose that it is enough to have a proper logical representation of this theory with a further theorem proving. However, the following problems arise when building such agents:

- **The transduction problem –** how to translate a real world into a proper logical symbolic description.

- **The representation/reasoning problem** – how to symbolically represent information about the real world entities and processes in a description language.

- **The reasoning problem** – how to get agents to reason with this information or symbolic descriptions in time for the results to be useful.

The former problem has led to work in areas such as vision, speech understanding, learning, while the latter led to works in knowledge representation, automated reasoning and automatic planning. Despite the huge number of developments and research, these problems are still quite far from being solved, which questions the efficiency of deliberative architectures compared to other approaches (such as the reactive architectures). The difficulties associated with the mentioned problems seem to be related with the complexity of commonsense reasoning, theorem proving and the symbol manipulation algorithms (Wooldridge and Jennings 1995).

Since the 1970s, the AI planning community has been involved in the design and development of artificial planning agents (predecessors of deliberative agents). This concern has remained for the next two decades. Wooldridge and Jennings claim that

"most innovations in agent design have come from this community" (Wooldridge and Jennings 1995). As a result, in 1971 STRIPS was created (Fikes and Nilsson 1971). STRIPS consisted in a simple planning system which tries to find a sequence of desired actions having in mind the symbolic descriptions of the world, the desired goal state and a set of actions. The STRIPS algorithm was very simple but needed further improvement, for it was unable to solve problems of even modern complexity. A lot of effort was made to surpass artificial planning limitations in time-constrained systems which resulted in the implementation of *hierarchical* (Sacerdoti 1974) and *non-linear* planning (Sacerdoti 1975). Despite this effort, the system proved to remain somewhat weak.

Only in late 1980s more successful attempts have been made in the design of planning agents. For instance, the IPEM (*Integrated Planning, Execution and Monitoring system*) presented an embedded sophisticated non-linear planner (Ambros-Ingerson and Steel 1988). Further, Wood's AUTODRIVE system simulated deliberative agents in a highly dynamic environment (a traffic simulation) (Wood 1993), Etzioni built 'softbots' that can plan and act in a UNIX environment (Etzioni, Lesh and Segal 1994) and Cohen's PHOENIX system was construed to simulate fire management (Cohen, et al. 1989).

Meanwhile, in 1976, Simon and Newel formulated the Physical Symbol Systems hypothesis (Newell and Simon 1976), stating that both human and artificial intelligence have the same principle – symbol representation and manipulation. Additionally, they claimed that there is only a quantitative and structural difference between the human's intelligence and the machine's intelligence, being the latter mush less complex.

In late 1980s and early 1990s some researches started to develop agent architectures based in the BDI (*Belief-Desire-Intention*) software model, which draws its inspiration from the philosophical theories of Bratman (Bratman 1988). He defended that intentions play a significant and distinct role in practical reasoning and cannot be reduced to beliefs and desires. In the BDI model, agent's beliefs about the world (its image of a world), desires (goals) and intentions are internally represented and practical reasoning is applied to decide which action to select. The first BDI-architecture was conceived in 1988, named IRMA (*Intelligent Resource-bounded Machine Architecture*) (Bratman, Israel and Pollack 1988). This architecture exemplifies that standard idea of a *deliberative agent*, as it is known today. It consists of four symbolic data structures: a plan library and representations of beliefs, desires and intentions. It also contains a *mean-end reasoner* to decide how to achieve an end using the available means; an *opportunity analyser* which generate further options for the agent; *a filtering process* to select the course of action compatible with the agent's intentions and commitments; a *deliberation process* to decide between competing options. Thus, it represents an agent embedding the symbolic representation and implementing the BDI. In 1990 this architecture has been evaluated in an experimental scenario denominated as the *Tileworld* (Pollack and Ringuette 1990). In 1995, Rao and Georgeff implemented a BDI-agent-oriented air-traffic management system, called OASIS (Rao and Georgeff 1995), integrating various aspects of BDI agent research. Other more modern BDI Agent Implementations: *dMARS* (Georgeff, Kinny and Wooldridge 2004), *JADEX* (Pokahr, Braubach and Lamersdor 2005), Agent Real-Time System (Vikhorev, Alechina and Logan 2009).

2.1.2 Reactive Architectures

The mentioned problems in 2.1.1 associated with symbolic AI and deliberative architectures led some researches to question the viability of the whole paradigm. Deliberative agents seem to be very ineffective in dynamic environment due to their inability to re-plan actions quickly enough and the high complexity of agents in even simple environments. In mid 1980s some researchers started to look to alternative techniques for building agents. As a result, the first **reactive architectures** started to appear. Stone and Veloso made a very clear comparative between reactive and deliberative agents (Stone and Veloso 2000):

- *"Reactive agents simply retrieve pre-set behaviors similar to reflexes without maintaining any internal state. On the other hand, deliberative agents behave more like they are thinking, by searching through a space of behaviors, maintaining internal state, and predicting the effects of actions."*

Thus, by the above statement we can retrieve two main features of the reactive architectures: they do not use complex symbolic reasoning; they do not include any kind of central symbolic world model (internal state). Other characteristics of the reactive control architectures: they make short-term predictions; tight sensor to actuator coupling; they divide the world in different 'situations' and each of them triggers one or more actions; they can be seen as the analogous of the reflexes in the human nervous system.

The first well-known reactive architecture was introduced in 1986 by Rodney Brooks and was called the *Subsumption Architecture* (R. Brooks 1986). He had become frustrated by the approaches made in control mechanism for autonomous robots so far and decided to create a new alternative for building agents, extending the view of AI. The subsumption architecture is a hierarchy of task-accomplishing behaviors, organized in layers (see Figure 2.1). Each layer has a control program capable of working at the speed of environmental change. Lower layers behaviors inhibit higher levels ones. Thus, the higher layers correspond to a more deliberative behavior, while the lower layers correspond to a more reactive and intuitive behavior. This way, the lowest layers can work like fast-adapting mechanisms, while the higher layers work to achieve the overall goal.

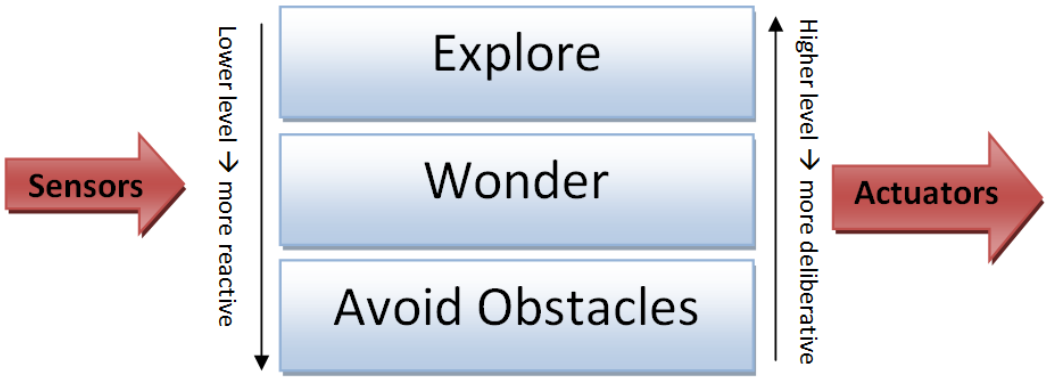


Figure 2.1: Subsumption Architecture **(R. Brooks 1986)**

The systems that are product of the subsumption architecture proved to be extremely simple. However, Brooks implemented this architecture in robots and verified that they are capable of doing tasks that would be impressive if they were accomplished by symbolic AI systems. A few years later, Steels described a simulation of subsumption-architecture agents in a "Mars explorer" system and reported that they could achieve near-optimal performance in certain tasks (Steels 1990).

In the following years, in more recent papers, (R. A. Brooks 1990), (R. A. Brooks 1991a) and (R. A. Brooks 1991b), Brooks put forward three main theses as a result of his work in the review of alternatives for the development of autonomous robots:

- Intelligent behavior can be generated without explicit representations of the kind that symbolic AI proposes.
- Intelligent behavior can be generated without explicit abstract reasoning of the kind that symbolic AI proposes.
- Intelligence is an emergent property of certain complex systems.

Brooks has also identified two key ideas in his research:

- **Situatedness and embodiment:** "Real" intelligence is situated in the world, not in disembodied systems such as theorem provers or expert systems.
- **Intelligence and emergence:** "Intelligent" behavior arises as a result of an agent's interaction with its environment. Also, intelligence is "in the eye of the beholder"; it is not an innate, isolated property.

There were other early researchers that studied alternatives to the symbolic AI model in late 1980s. In 1986, Chapman and his co-worker Agre also reported the theoretical difficulties with the symbolic planning in (Chapman and Agre 1986) and developed the PENGI system in 1987 (Agre and Chapman 1987). PENGI is a simulated computer game in which the characters are controlled by a new architecture based in "routines"

tasks with periodic updating to handle new kinds of problems. This main idea was observed by Agre who claimed that most everyday activity is "routine" and that it requires little – if any – new abstract reasoning.

Nowadays, we can find several models of behavior for reactive control architectures. The more intuitive are the ***productions* control architectures**. *Productions* consist of condition-action rules in the form: **if** condition **then** action. Normally, the conditions are Boolean and are triggered by external events percepted by the agent. Actions can be either external (e.g., pick an object from the ground) or internal (e.g., write something into the internal memory) and they can be either performed or not. *Productions rules* can be organized in flat structures or other more complex structures such as trees or hierarchy. The referred subsumption architecture is an example of a *production* control architecture with layers of interconnected behaviors organized into a simple stack. Another example is the PENGI system described above.

Another approach for modeling reactive architectures is the **finite state machine**. They specify the behavior of an agent trough a model consisted of states and transitions between these states (see Figure 2.2). The transitions are *productions* rules in the form of **if** condition **then** activate-new-state. In every instant, only one state can be active and each state can be associated with an act or a script. An act is an atomic action that should be performed by the agent while a script describes a sequence of actions. A script can be broke down to several scripts and in this way we can exploit the hierarchical finite state machine in which every state can contain substates. Despite the fact that the first theoretical studies in finite state machines and automata theory date back to the 1960s (Booth 1967), they were only first applied to reactive intelligent agents in late 1980s. One of the earlier works belongs to Rosenschein and Kaelbling that specified the *situated automata* paradigm with their works between the years 1985 and 1991. In this paradigm, an agent is specified in declarative terms and then compiled to a digital machine (Kaelbling and Rosenschein 1990). This approach has attracted much interest, as it appears to combine the best elements of reactive and symbolic systems. These days, the finite state machines are applied to physical autonomous robots and to softbots. In the paper of Damian Isla, (Isla 2005) we can find a description of computer game bots, implemented in the video game *Halo 2*, which use hierarchical finite state machines.

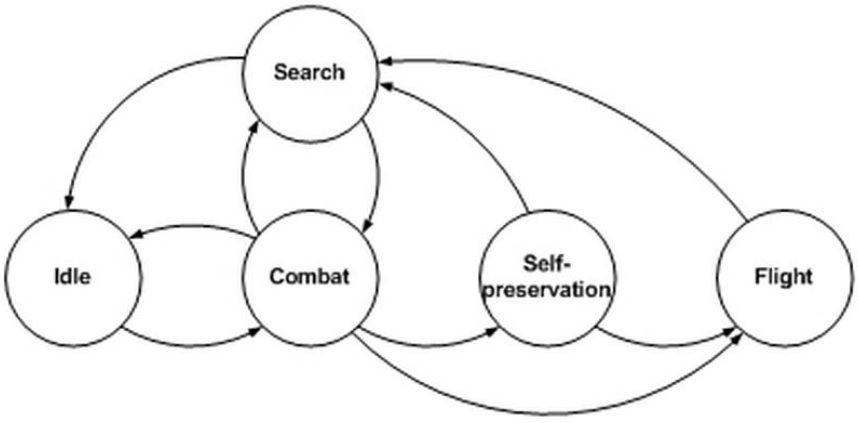


Figure 2.2: A Finite State Machine used in softbots in *Halo 2* **(Isla 2005)**

The referred approaches for reactive architectures can be combined with **fuzzy logic**. In these cases, conditions, states and actions are no more Boolean but are approximate and smooth. Thus, the resulted behavior will reflect smoother transitions, especially in the case of transitions between two tasks. Fuzzy logic techniques have been used for the implementation of several intelligent controllers and for navigation in autonomous vehicles - see (Driankov and Saffiotti 2001).

We can also find reactive architectures based in **connectionist networks**. Connectionist networks are composed of unit with inputs links (that feed the unit with "an abstract activity") and outputs links that propagate the activity for the subsequent units. The associated architectures have several advantages over the other referred reactive architectures, since they proved to provide an even smoother behavior of an agent and they are often adaptive. They also have some flaws: it is more complicated to extract knowledge regarding the behavior of an agent using connectionist networks that using *production* rules; if we want to exploit the adaptive feature, only relative simple behavior is proved to be effective. The most famous connectionist networks are the *artificial neural networks* whose studies started to appear in mid 1950s (see Figure 2.3). Currently, they are very used in Machine Learning and Data Mining to build classifiers or for pattern recognition. Their appliances have also extended to Computer Vision, as described in (Egmont-Petersen, Ridder and D. 2002), and in the field of robotics to design direct manipulators of industrial robots or to design controllers for steering and path-planning of autonomous robot vehicles - see (Markoski, et al. 2009).

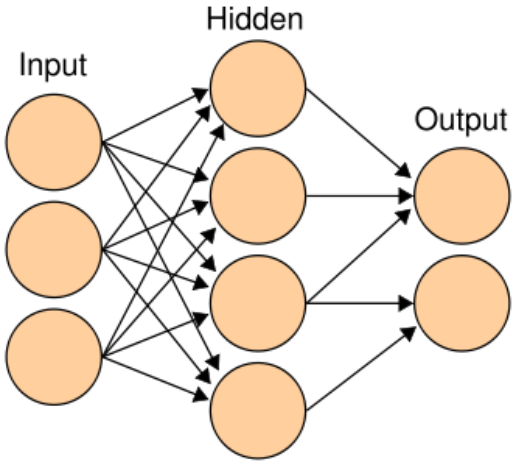


Figure 2.3: An Artificial Neural Network (ANN)

Other important reactive control architectures are the ones based in pure **classic control theory**. They specify controllers that manipulate the inputs of a system to obtain the desired effect on the output of the system. This is accomplished using a feedback control loop, in which a controller periodically percepts the outputs of a system, measures the error to the reference and responds to the system in order to compensate and/or minimize the error (see Figure 2.4).

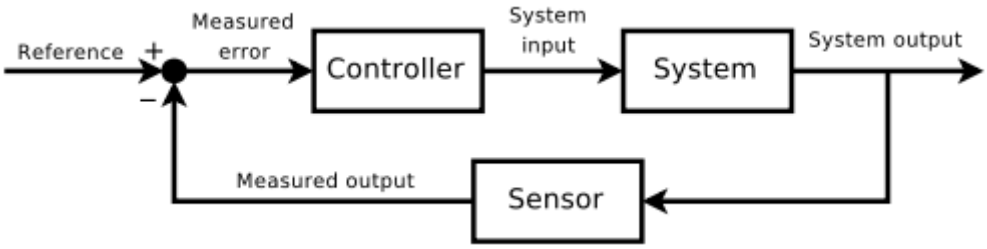


Figure 2.4: Feedback control loop

This type of control is very used in industrial processes. The most-used feedback control design is the PID controller. PID is an acronym for "*Proportional-Integral-Differential*". The PID controller includes elements with those three functions to minimize its error. Akira referred that *"a study made in 1989 in Japan indicated that more than 90% of the controllers used in process industries are PID controllers and advanced versions of the PID controller."* (Araki 2000). Nowadays, PID controllers are still used in several industrial processes and in the robotic field, such as in the design of steering control for autonomous mobile robots.

Despite the several approaches made for the reactive architectures and their application in the design of effective intelligent agents, they still present some limitations. Reactive agents without environment models must have sufficient information of the local environment. Having this in mind, it is clear that they have a short-term view of the goal. Previously, it was referred that some reactive agents can have adaptive capabilities (e.g. if we use an artificial neural network). However, it is difficult to make reactive agents that learn, since they have no internal memory that could be exploited by the controller. The building of these agents is mainly based on test-fault approach which implies a long and laborious process. Another general problem regarding reactive agents is that it is hard to engineer agents capable of a large number of behaviors since the dynamics of the interactions become too complex to understand.

2.1.3 Hybrid Architectures

Having in mind the problems already mentioned, associated with both deliberative and reactive architectures, it is with no surprise that many researchers suggested that neither a completely deliberative nor completely reactive approach is suitable for building agents. They argued the case of **hybrid architectures**, which attempt to marry classical and alternative approaches with the hopes of enjoying with the "best of two worlds". Most of the hybrid approaches specify an agent built out of two (or more) subsystems:

- A **deliberative subsystem**, which contains a symbolic world model and is capable of elaborating plans and making decisions;

- A **reactive subsystems**, which is capable of reacting rapidly to events without complex reasoning;

These subsystems are often organized in layers, in which the bottom layer has no symbolic representation, while the upper layers do employ such representation. An important aspect of these architectures is the control framework which manages the interactions between the various layers. In these architectures, we can have different types of layering (see Figure 2.5).

- **Horizontal Layering** – all layers are connected to the inputs and the outputs. The mediator (overall control function) is needed to determine which action to take
- **Vertical Layering** – the input are connected sequentially i.e., the input is connected to one layer, which is connected to the next, and so on. The last layer is connected to the outputs. This layering can be divided in two subtypes: *one pass* and *two pass*. In *one pass*, the perceptions and actions are passed up, while in the *two pass* they also bounce down.

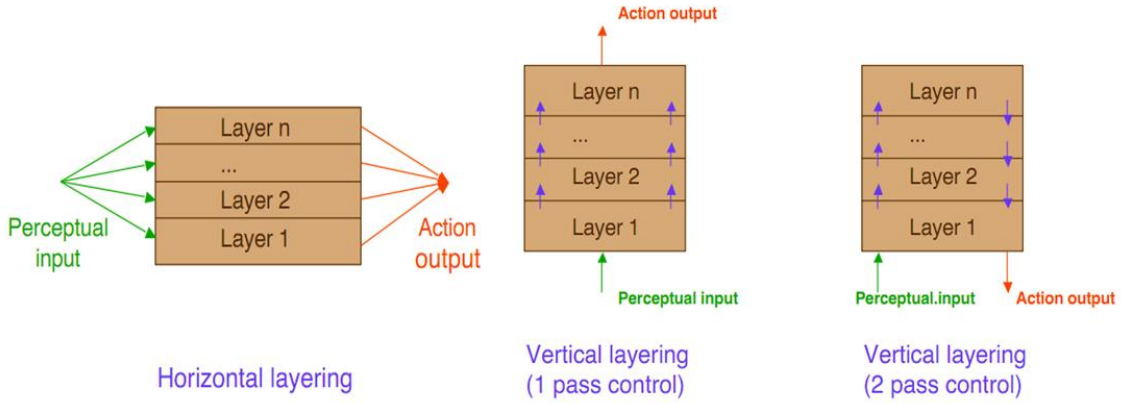


Figure 2.5: Different types of layering in Hybrid Systems **(Munneke, Wahlstrom and Zaccara 1998)**

The *Procedural Reasoning System* (PRS), developed by Georgeff and Lansky in 1987 (Georgeff and Lansky 1987), is one of best-known hybrid architectures for agents. It uses BDI model (see Deliberative Architectures) and is constituted by a set of *knowledge areas* (KA), each of which is associated with an *invocation condition* that can activate it. KAs can be activated in a goal-driven or data-driven fashion, but they also can be reactive. This allows the PRS to respond rapidly to sudden change in the environment. PRS was applied as a fault detector system on the NASA space shuttle (Georgeff and Ingrand 1990).

Another well-known hybrid agent architecture is the *TouringMachines* developed by Fergunson for his 1992 PhD thesis (Ferguson 1992). This architecture is a clear example of horizontal layering (see Figure 2.6). This architecture consists of two subsystems (perception and action), which interact directly with the agent's environment, and three control layers, embedded in a control framework, which mediates between the layers. The *reactive layer* is implemented as a set of *production*

(situation-action) rules, like in the subsumption architecture. The *planning layer* build plans and selects actions to execute in order to achieve the agent's goals. The *modeling layer* contains symbolic representations of the "cognitive state" of other entities in the agent's environment. It also selects new goals for the *planning layer*. These three layers are capable of intercommunication (via message passing) and are embedded in a control framework that deals with conflicting action proposals from the different layers. The control framework achieves this using *control rules*.

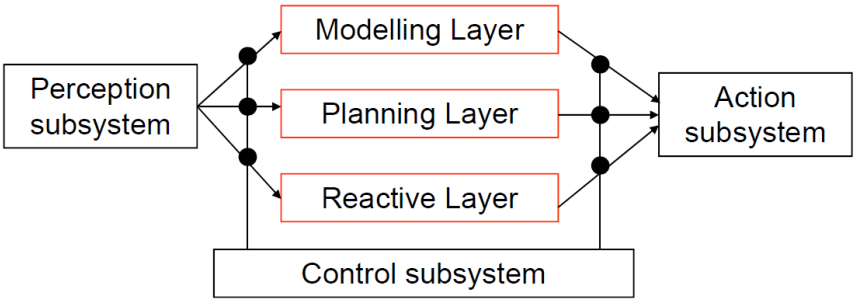


Figure 2.6: The *TouringMachines* architecture (Ferguson 1992)

An example of vertical layering is the architecture INTERRAP, designed by Müller in 1994 (Müller 1994). In this hybrid architecture, each successive layer represents a higher level of abstraction than the one below it (see Figure 2.7). These layers are then subdivided in two vertical layers: one containing control components and the other knowledge bases. The reactive capability of this architecture is made possible by the Behavior-Based Layer which manipulates a set of *patterns of behavior*. At higher layers, we can find components responsible for the planning and the deliberative behavior, using a planning library and a social knowledge data base. Thus, INTERRAP is both data and goal driven and, as result of changes to the world model, various patterns of behavior may be activated, dropped and executed.

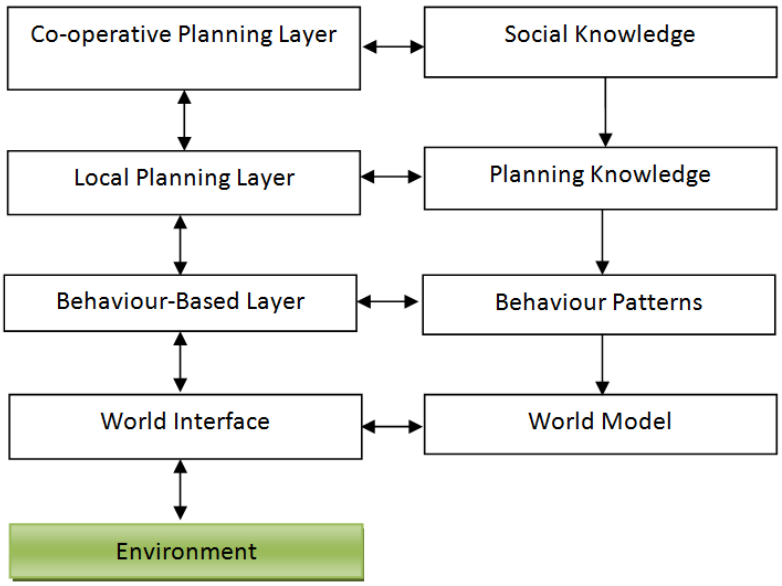


Figure 2.7: INTERRAP Hybrid Architecture (Müller 1994)

In 2001, Nilsson described an hybrid architecture for linking perception and action in a robot, which consisted of three hierarchical towers - perception, action and model (see Figure 2.8). As a result, he specified the Triple Tower Architecture (Nilsson 2001). This architecture contains the following towers:

- ***Perception Tower*** – *"consists of rules that increasingly abstract descriptions of the situation starting with the primitive predicates produced by the robot's sensory apparatus." (Nilsson 2001)*

- ***Model Tower*** – "continuously keeps the descriptions faithful to the current environmental situation by a 'truth-maintenance' system." *(Nilsson 2001).*

- ***Action Tower*** – ***"****consists of a loose hierarchy of action routines that are triggered by the contents of the model tower. The lowest level action routines are simple reflexes---evoked by predicates corresponding to primitive percepts. More complex actions are evoked by more abstract predicates appropriate for those actions. High-level actions "call" other actions until the process bottoms out at the primitive actions that actually affect the environment." (*Nilsson *2001).*

Nilsson illustrated the operations of this architecture describing a triple-tower system for building tower of blocks on a table.

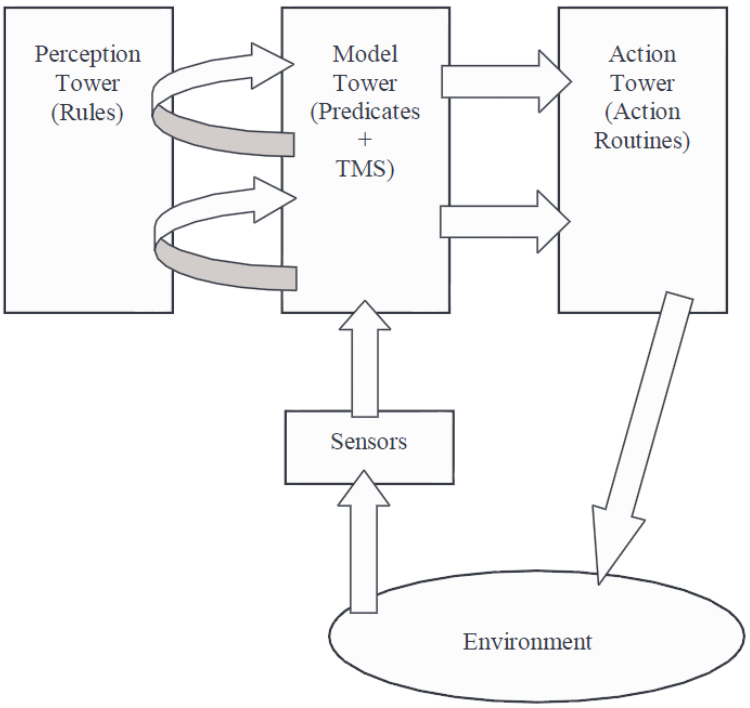


Figure 2.8: Triple-Tower Architecture (Nilsson 2001)

Other well-known hybrid architectures applied to robotic control are: the DAMN (Rosenblatt 1997) and the DD&P (Hertzberg and Schönherr 2001). The DAMN consist of a Distributed Architecture for Mobile Navigation that contains a set of weighted modules that run concurrently and send votes to be *combined*. These modules represent various behaviors (such as obstacle avoidance and path planning) and have various working speed. Despite being an architecture capable of both deliberative and reactive behavior, it suffers from being very specialized to navigation. For more information, see (Rosenblatt 1997). The DD&P architecture is also constituted by modules, in which behaviors are leveled and interact through shared variables. However, its planning components can affect any behavior in any level. It also has components that permanently update the propositional world including the goal and monitors the state of the current plan. For more information regarding the DD&P, see (Hertzberg and Schönherr 2001). Another interesting hybrid architecture is the one described in (Secchi, et al. 1999). The proposed architecture is behavior-based but combines aspects of classic control. It was implemented in a mobile robot which travelled in semi-structured environments. It also contained a priority scheme based in Petri nets to activate the different possible behaviors in the robot.

Currently, hybrid architectures are a very active area of work, with proven advantages over both purely deliberative and purely reactive architectures. However, one potential difficulty of these architectures is that they lack of deep theory formalization. Wooldridge and Jennings stated that *"It is a matter of debate whether this need be considered a serious disadvantage, but one argument is that unless we have a good theoretical model of a particular agent or agent architecture, then we shall never really*

understand why it works. This is likely to make it difficult to generalize and reproduce results in varying domains." (Wooldridge and Jennings 1995)

2.2 Cognitive Architectures

All agent architectures can be considered cognitive architectures if we interpret that agent architectures intend to implement *cognitive processes* to intelligent agents, such as reasoning, reflexes (rapid response to an environmental change) or internal representation of the world knowledge. However, we are going to refer to cognitive architectures accounting for a different perspective. Therefore, throughout this document, *cognitive architectures* are going to be referred to architectures that propose computational processes that act like certain cognitive systems (most often, like a person). This way, cognitive architectures form a subset of agent architectures and consist in an interdisciplinary research area in which converge the fields of artificial intelligence, cognitive psychology/cognitive science, neuroscience and philosophy of mind. Researchers of cognitive architectures believe that the understanding of (human, animal or machine) cognitive processes means being able to implement them in a working system.

In the next sections we are going to present different types of classifications of cognitive architectures and analyze three of the historically most representative cognitive architectures: RCS, Soar and ACT-R. These architectures are still active lines of research.

2.2.1 Classification of Cognitive Architectures

We can analyze cognitive architectures in two dimensions: according to their main purpose (or stream) or according to their general structural paradigm. This leads to different types of classifications.

If we classify cognitive architectures according to their purpose, we can distinguish three main categories:

- ***Architectures that model human cognition.*** *One of the main interests of cognitive science is to produce a complete theory of human mind integrating all the partial models, such as models of memory, vision or learning. These architectures are based upon data and experiments from psychology or neurophysiology, and tested upon new breakthroughs. However, these architectures do not limit themselves to be theoretical models, and have also practical applications. An example of this is the cognitive architecture ACT-R which is applied in software based learning systems: the Cognitive Tutors for Mathematics that is used in thousands of schools across the United States - for more information, see (Anderson e Gluck 2001). Examples of these architectures: ACT-R and Atlantis.*

- ***Architectures that model general intelligence***. *These architectures are closely related to the first ones but, despite of also being based upon the human mind (as the only agreed intelligent system up to date), they do not constraint to explain the human mind in its actual physiological implementation. They address the subject of general intelligent agents, mainly from a problem-solving based perspective. Examples of these architectures: Soar and BB1.*

- ***Architectures to develop intelligent control systems.*** *These architectures have a more engineering perspective and relate to those addressing the general intelligence problem, but with the main goal of applying it to real technical systems. They are intended as more powerful controllers for systems in real environments, and are mainly applied in robotics, namely in UGV (Unmanned Ground Vehicle). An example of these architectures: RCS and ICARUS.*

In this work, we are especially interested in the last type, since the long-term aim is to apply cognitive architectures to enhance the autonomy and robustness of intelligent agents and due to the context of the work of this document: autonomous navigation.

Cognitive architectures can also be divided in the following main paradigm approaches which reflect in the structure of the subsequent intelligent systems:

- **Symbolic or the computational approach** – this approach is based on the mind-is-like-a-computer analogy. Thus, in this approach, the human mind is best conceived as an information processing system very similar to a digital computer. The resultant architectures are based on a set of generic rules and are very similar to deliberative architectures (see 2.1.1 - Deliberative Architectures). An example of this approach is the Soar architecture.

- **Connectionist approach –** this approach is based on the principle that the mental phenomena can be explained by the emergent properties of processing units. Thus, this approach relies on the belief that intelligent behavior can emerge from the specification of a set of simple units and their interaction. The form of the connections and the types of units can vary from model to model. The most well-known type of representation of this approach is as Artificial Neural Networks (already mentioned in 2.1.2 - Reactive Architectures).

- **Hybrid approach –** this is an explicit approach which defines architectures with a combination of the above mentioned paradigms and respective types of processing. An example of this approach is the ACT-R architecture, with its symbolic and subsymbolic (connectionist) levels.

It is remarkable that the symbolic paradigm is strongly related to deliberative architectures, while the connectionist is related to the reactive approach. Another ways of distinction of the cognitive architecture is whether they are centralized or distributive or if their systems should be design in a top-down or bottom-up perspective.

2.2.2 RCS

RCS (Albus and Barbera 2004) stands for *Real-time Control System* being a Reference Model Architecture for building intelligent control systems and developed by the NIST (National Institute of Standards and Technology). RCS have been applied in several domains, such as in UGV (Unmanned Ground Vehicles) (Albus, et al. 2002), space robotics (NASREM) or manufacturing (ISAM).

This architecture is based in the functional decomposition of the system in nodes, as fundamental units of control, which are organized in different hierarchy-levels. The different levels of the hierarchy represent *different levels of resolution*, i.e., different abstraction levels. The lower level in the RCS hierarchy is connected to sensors and actuators of the system. The nodes are interconnected both vertically through the levels and horizontally within the same level via a communication system. The hierarchy levels and the nodes are immutable, but their connection can change according to the mission, forming a "command tree". Each node possesses four modules that correspond to different cognitive processes:

- **Sensory Processing module (SP):** performs several perception functions that enables the node to update the world model and processes the sensory input stream to be sent to the upper nodes.

- **World Modeling module (WM) and Knowledge Database (KD):** this is where the node stores its knowledge of the world in models composed by objects and events. The WM runs the models so as to make predictions and simulations that operate the perceptual processes and enable planning in the behavior generation node.

- **Behavior Generation module (BG):** receives instructions from the upper nodes and is responsible of the planning and action tasks, which may include sending the instructions to lower nodes.

- **Value Judgment module (VJ):** computes value, determines importance, assesses reliability and generates reward and punishment.

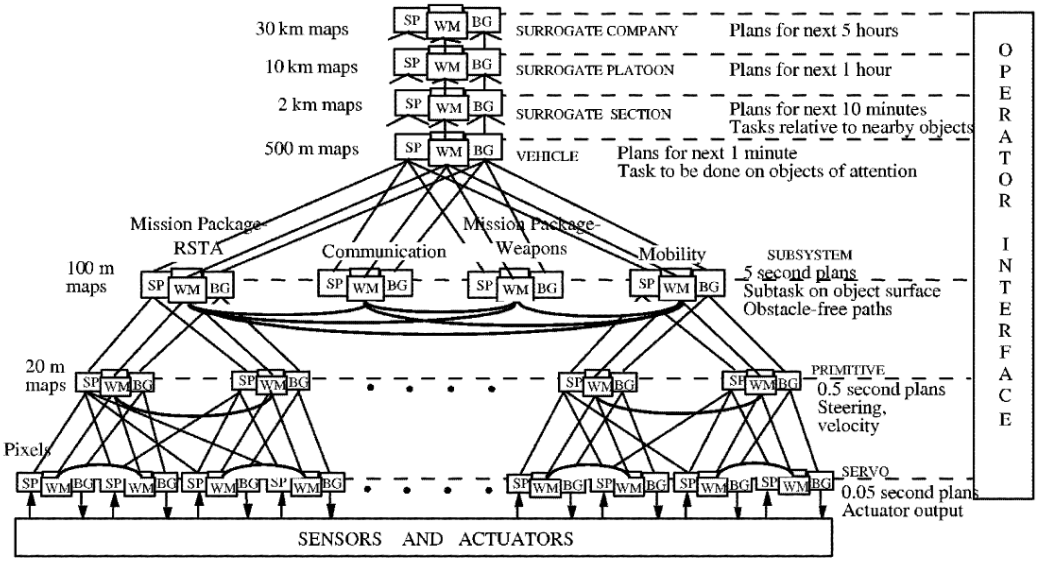


Figure 2.9: Example of a RCS hierarchy **(Albus, et al. 2002)**

RCS adequately encapsulates the cognitive processes through the nodes and presents certain adaptability, since it enables to change the connections of the nodes for each different mission. Nevertheless, the fact that the nodes and the hierarchical level organization are static brings some constraint to the architecture. It also lacks of models of the internal architectures, meaning that the nodes do not have models of themselves or of the neighboring nodes in the same way that they have models of the environment and of the physical system that controls the architecture. Therefore, it lacks of self-awareness mechanism.

2.2.3 Soar

Soar (*State, Operator And Result*) was based in the Newell's idea for the development of intelligent agents and theories of cognition (Laird, Newell and Rosenbloom 1987). Soar is a symbolic general cognitive architecture and it is designed based on the hypothesis that all deliberative goal-oriented behavior can be cast as the selection and application of *operators* to a *state*. A *state* is a representation of the current problem-solving situation; an *operator* transforms a state (makes changes to the representation); and a *goal* is a desired outcome of the problem-solving activity.

Soar is constituted of three memories and I/O interfaces which interact in every execution cycle:

- **Input/output functions:** at the beginning of each cycle, they update the objects that are in the Soar's working memory. At the end of each cycle, they embody the required actions for to the operations stored in the working memory.

- **Working memory:** it stores the system's knowledge of the current situation as a set of working memory elements (WME) which consist of an identifier-attribute-value. All WME's sharing its identifier are an *object*, which can stand

for data from sensors, results of intermediate inference, active goals, operator or any other entity of the problem. The operators are labeled with preferences and in each cycle, only one operator can be applied.

- **Production memory:** contains the productions, which consist of a set of conditions and a set of actions. In each cycle, the production *fires* the actions of which conditions are in agree with the working memory's state. The conditions of a production typically refer to presence or absence of objects in working memory. Production actions can be one of the following: object proposal, operator comparison, operator application or state elaboration.

- **Preference memory:** stores preferences which determine the selection of the current operator. Preferences are suggestions or imperatives about the current operator, or information about how suggested operators compare to the others.

If an execution cycle cannot solve the operator selection, an impasse happens and Soar automatically generates a new substate which consists in the solving of the impasse. In this way, a decomposition of substates is made. While the successive substates are solved, Soar stores the process as new productions. This is the main learning mechanism in Soar, called *chunking*.

Soar architecture addresses mainly the deliberative part of a cognitive system, not providing fine grained design patterns for perception and grounding. Besides, there is no modeling of the functioning of Soar's cognitive operation, so the architecture is lacking of self-awareness or consciousness. Nevertheless, it presents several solutions of potential interest: a homogeneous knowledge management in the *productions* and in the objects in the memory, which enables meta-knowledge and a learning mechanism to enhance the procedural knowledge. The *chunking* mechanism provides a great benefit to the architecture in autonomy and adaptability.

2.2.4 ACT-R

ACT-R (*Adaptive Control of Thought - Rational*) is a cognitive architecture mainly developed by J. Anderson in Carnegie Mellon that is also a theory about how human cognition works. Most of the ACT-R basic assumptions are inspired by the progresses of cognitive neuroscience and, in fact, ACT-R can be seen and described as a specification of how the brain itself is organized in a way that enables individual processing modules to produce cognition (Lebiere e Anderson 1993). It can also be seen as a programming language to build models of cognitive processes or to elaborate different tasks, such as puzzle solving or aircraft piloting.

Like other influential cognitive architectures, the ACT-R is based in expert system's mechanisms. It is structured in opaque modules that interact trough buffers. The buffers work as interfaces and only store one element in memory at a time. The contents of the buffers at a given moment in time represent the state of ACT-R at that moment.

There are four types of modules in ACT-R:

- **Perceptual-motor modules:** which take care of the interface with the real world (i.e., with a simulation of the real world). The most well-developed perceptual-motor modules in ACT-R are the visual and the motor modules.

- **Memory modules:** there is a declarative memory, consisting of facts and homogeneous knowledge, encoded in units called *chunks*. There is also a procedural memory for making productions. Productions represent knowledge about how we do things: for instance, knowledge about how to write the letter 'a', about how to drive a car, or about how to perform addition.

- **Intentional modules:** defines the system's main goals.

- **Pattern Matcher:** searches for a production that matches the current state of the buffers. Only one production can be executed at a given moment. That production, when executed, can modify the buffers and thus change the state of the system. Thus, in ACT-R cognition unfolds as a succession of production firings.

Besides having a periodical symbolical operation, ACT-R also presents subsymbolic mechanisms, consisting of equations that control the activation of production and equations that can retrieve *chunks* from the declarative memory. Besides, the mechanism in the perceptual-motor modules can also be subsymbolic, which makes ACT-R an hybrid architecture.

ACT-R is, as Soar, an architecture centered in the conceptual operations. Despite having perceptual modules and grounding, it has been developed as a model of the human mind mechanism, which makes it hard to apply to technical systems. Compared to Soar, it has the advantage of being hybrid, which can be reflected in the concurrency of the module's processes. Nevertheless, it has the disadvantage of dealing with implicit knowledge. ACT-R presents learning mechanism in the symbolic level (with the creation of productions) and in the subsymbolic level (with the adjustment of activations). However, its learning mechanism revealed to be less powerful than the *chunking* in Soar.

2.3 Fault-tolerance in Autonomous Systems

- *"The more the number of components, the more things there are that could be faulty. (…) Similarly, the more complex a component, the more chance there are of it being faulty." (Jalote 1994)*

Many complex systems, such as the robotic control systems, have to be able to achieve its objectives under data noise and uncertainty. Eventually, part of the system may be damaged or malfunction during operation, compromising system cohesion and therefore its capacity to achieve objectives. Fault-tolerance techniques have been developed to provide the system with mechanisms to react to these circumstances by adapting itself. According to (Jalote 1994) , fault tolerant systems evaluate self-performance in terms of *dependability*, which is defined as the trustworthiness of a system such that reliance can justifiably be placed on the service it delivers. The most significant attributes of *dependability* are *reliability*, *availability*, *safety* and *security*.

Jalote distinguished three concepts in relation with reliability: a *failure* is a deviation of the system behavior from the specification. An *error* is the part of the system which leads to that failure. Finally, a *fault* is the cause of an error. A fault can be an internal event in the system, a change in the environmental conditions or it can even be a wrong control action given by a human operator or an error in the design of the system.

There are two main general approaches to improve the reliability of a system: *fault prevention* and *fault tolerance*. However, it is assumed that fault prevention techniques will never be able to eliminate all possible faults, as any real-time system is likely to have or develop fault in it. Therefore, in order to increase the reliability of a system, fault tolerance techniques were proven to be more effective, in which systems can provide the service in spite of the existence of faults.

In artificial systems, fault-tolerance is usually implemented in four phases (Jalote 1994):

- **Error detection:** The presence of a fault is deduced by detecting an error in the state of the subsystem.

- **Damage confinement and assessment:** the damage caused by a fault is evaluated and delimited (affected parts are identified and effect on objectives estimated).

- **Error recovery:** correction of the error to avoid its propagation.

- **Fault treatment and continued service:** faulty parts of the system are deactivated or reconfigured and the system continues operation

Fault tolerance can be, and is applied at various levels in a computer system. Therefore, it distinguishes between hardware and software. Hardware fault tolerance is based on fault and error models, which permit identifying faults by the appearance of their effects at higher layers in the system (software layers). Hardware fault tolerance can be implemented by several techniques, being the most known:

- **TMR (Triple Modular Redundancy):** three hardware clones operate in parallel and vote for a solution.

- **Dynamic redundancy:** spare, redundant components to be used if the normal one fails.

- **Coding:** addition of check bits to the information bits such that errors in some bits can be detected and, if possible, corrected.

Software fault tolerance can be based on a *physical model* of the system, which describes the actual subsystems and their connections, or on a *logical model*, which describes the system from the point of view of processing. In general, software fault tolerance is based on the following fault classification:

- **Crash fault:** fault cause component to halt or to lose its internal state.

- **Omission fault:** caused the component to no respond to certain inputs.

- **Timing/Performance fault:** the response of the component is too early or too late.

- **Byzantine fault:** arbitrary fault causing arbitrary behavior of the component.

A model for fault-tolerant systems that influenced the work associated with this document is the one described in (Blanke, et al. 2006). This model is based on analytical redundancy, opposing to physical redundancy (a dynamic redundancy of physical components). The authors defend that the industry cannot afford to use physical redundancy in large scale, since the duplication of each physical component is non-viable in most cases. Therefore, in the proposed model, an explicit mathematical model is used to perform the two steps of fault-tolerant control. The fault is diagnosed by using information included in the model and in the on-line measurement signal. Then the model is adapted to the faulty situation and the controller is re-designed so that the closed-loop system including the faulty plant satisfies the given specification.

The architecture of fault tolerant control according to (Blanke, et al. 2006) can be seen in Figure 2.10. The diagnosis block uses the measured input and output and tests their consistency with the plant model. Its result is a characterization of the fault with sufficient accuracy for the controller re-design. The diagnostic result f is assumed to be identical to the fault f occurring in the system. The re-design block uses the fault information and adjusts the controller to the faulty situation.

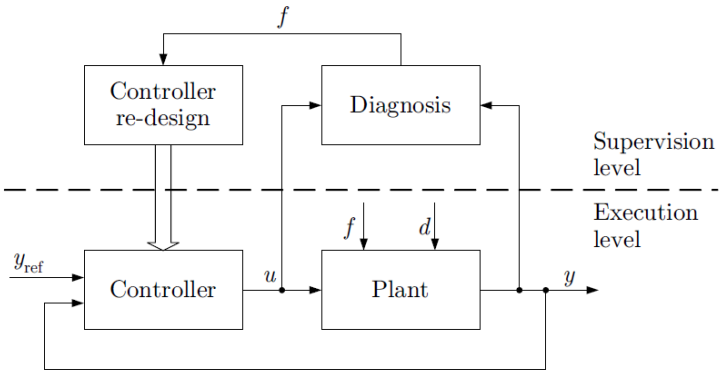


Figure 2.10: Architecture of fault-tolerant control as described in (Blanke, et al. 2006)

The two principal ways of controller re-design are:

- ***Fault accommodation****:* when a fail occurs, the controller re-designer adapts the controller parameters to the dynamical properties of the faulty plat.

- ***Control reconfiguration:*** if fault accommodation is impossible, the complete control loop has to be reconfigured. Reconfiguration includes the selection of a new control configuration where alternative input and output signals are used.

The most relevant definition in (Blanke, et al. 2006) is the Generic component model, in which each component of the fault tolerant architecture is composed, among other things, by its *state transition graph*, by the *services* it provides and the *version* of the respective services. According to this model, a *component* is more associated with a specific service of the system than with the physical/logical component that provides that service. For example, in a system for an autonomous mobile robot, there can be can be a *localization component* that provides the *service* of *localization*. This *service* can exist under different *versions*, each of which can use different physical components, such as a laser, cameras, GPS, etc. Thus, in case of physical failures, the fault tolerant control must maintain the *component* and the *service* but it can switch the version of the *service*.

2.3.1 Fault-tolerance in Robotics

In robotic programming, the interaction with the world and its unpredictability make the problem of error tolerance/recovery especially important. It becomes crucial in situations where humans operators cannot manually repair or provide compensation for damage or failure. Thus, some research work in fault tolerance in robotics has been done in the past years, being currently a very active field of study.

In 1978, Srinivas attacked the problem of fault recovery, with the intention of designing a robot that could recover intelligently form failures that occur during the execution of tasks in a static world with no other agents of change (Srinivas 1978). As a result, he

implemented a computer program called MEND, which automated recovery from failure in simple manipulation tasks in an autonomous mobile robot. Compared to other related approaches from that time, MEND had the special characteristic of trying to find the explanation of the failure through logical reasoning, in order to determine where the problem lies and to execute a proper recovery plan. However, this system presented several limitations, which derived from the extensive use of plan formation as the basis for constructing robot programs and on the choice of checking only the preconditions of the actions. In this way, an error may be discovered later than when it appeared.

In 1983, M. Gini and G. Gini, proposed a framework for error recovery in robots programs, which included a monitor program that identified the appearance of any error and attempt to correct that error (Gini and Giuseppina 1983). The recovery procedure was activated by the identification of the error, and the proper recovery procedure is detected using information extracted from a knowledge base. The knowledge base contained rules about correction activities and about interpretation of the sensor data. The dynamic model of the program included the *Initial Model*, the *Expected Model* and the *Current Model*. Error identification is possible comparing the *Current Model* with the *Expected Model*. The monitor controlled both the preconditions before executing any instruction and the postconditions at the end, using the first check as protective measure, since it should not be needed if the program does not have logical errors. Though the examples presented in the work are simple and limited, they authors defended that they defined a general framework for error recovery in robots programs, which could be extended and applied in several domains.

In 2010, Bongard and Lipson stated that the Srinivas approach on error diagnosis and recovery, as well as many subsequent approaches (such as the refereed above) were somehow ineffective, since they required exclusively online operation (repeating testing on the physical robot) and could not handle unanticipated errors. By the time, there were already some developments of offline error diagnostic and recovery systems, such as the one described in (Baydar and Saitou. 2001) which relies on Bayesian inference for error diagnosis. Nevertheless, Bongard and Lipson pointed out their limitations because of the large number of hardware trials needed to recover from an error (as many as 400 in many cases). Therefore, in the paper (Bongard and Lipson 2004) they introduced a two-stage evolutionary algorithm which can automatically diagnosis and recover from a wide range of unanticipated internal damage or external environmental change using only four hardware trials.

In 2010, the first fault-tolerant multi-robot control architecture for sensory-based coverage is proposed (Ozkan, et al. 2010). According to the authors, many works have been made to increase the efficiency of multi-robot coverage problem, but none have considered robot failures. They proposed a fault-control control architecture that they implemented in P3-DX robots in laboratory and in *MobileSim* simulation environments. Robot failures are detected using the heartbeat strategy that does not require time synchronization between robots. The proposed architecture is distributed in terms of fault detection, having a modular structure.

2.4 Autonomous Mobile Robotics Techniques

Since all the implementations associated with the work of this document are going to be made on a mobile robot, it is important to refer some background and state of the art of the techniques used for navigation in autonomous mobile robots. This sub-chapter is crucial for a better understanding of the underlying methods used beneath the core system.

2.4.1 SLAM

SLAM (Simultaneous Localization and Mapping) is a technique used in mobile robotics in which a robot builds a map of an unknown environment, keeping at the same time track of its localization in this environment. It was original developed in (Leonard e Durrant-Whyte 1991).

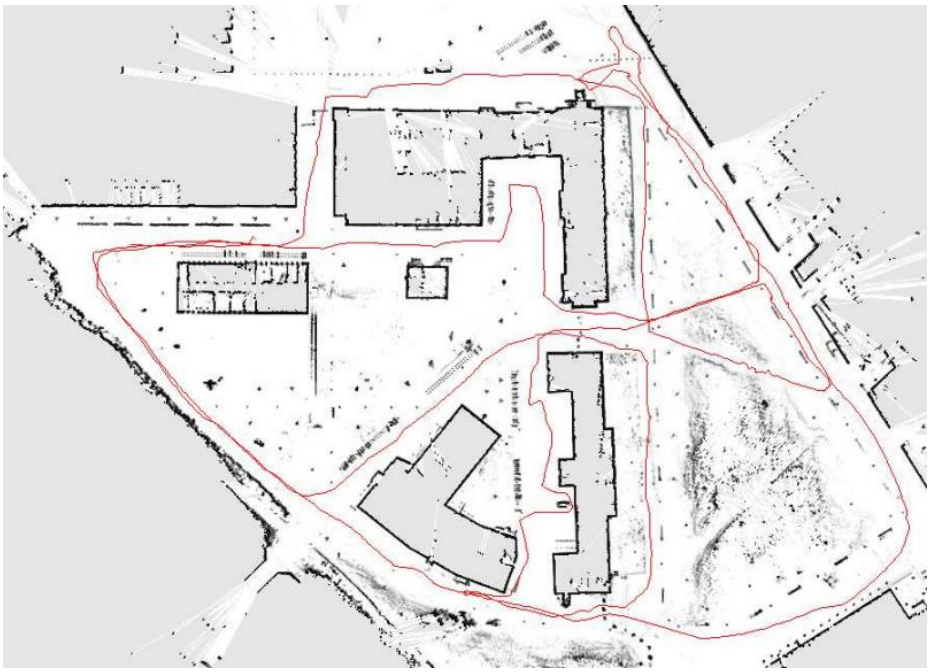


Figure 2.11: Map generated by SLAM (Grisetti, Stachniss e Burgard 2006)

One of the main problems for autonomous mobile robots is keeping track of their localization accurately. It is said that the SLAM consists in making the robot to answer two questions:

- ***Where am I?*:** an accurate localization is not possible without having references of the environment, a map that holds all the obstacles representations that the robot could find along the way and the landmarks that the robot will use to locate itself accurately.

- ***What does the world look like?:*** Mapping is the problem of integrating the information gathered by a set of sensors into a consistent model and depicting that information as a given representation. Central aspects in mapping are the representation of the environment and the interpretation of sensor data.

The need to use landmarks for location estimation derives from the fact that the robot's odometry, primary source for self-localization, is often erroneous (see Figure 2.12). Thus the robot needs to use more data, such as the scans obtained by a laser, to estimate its position in the world.

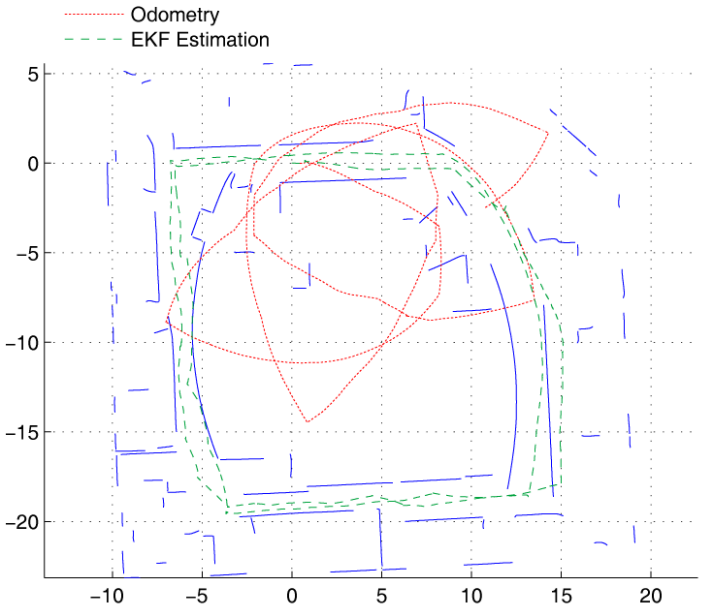


Figure 2.12: Trajectories estimated by a robot. In red we can see the trajectory estimated by the odometry (Sánchez 2009)

The solution to the SLAM problem is considered the main key to the autonomy of mobile robots and there are many efforts made by many researchers to push forward the developments of this technique.

The main difficulties associated with SLAM are the following:

- The observations made by the robot are referred to its own reference system that is affected by errors in the odometry. Thus, we add observation's imprecision to inaccuracy in the location.

- For bigger maps, the odometry errors are bigger, as well as the computational cost.

- The environments are usually dynamic, especially for domestic or guide robots. Thus, in many cases, the assumed fixed references can be "hidden" by some new elements in the environment, making the task of localization even more difficult.

- In some environments, there may be very similar landmarks, which can "confuse" the SLAM to make wrong associations.

- Most of the today's SLAM implementations do not use 3-D information. The sensing is usually made in a horizontal plane.

There are three base SLAM algorithms from which, currently, almost all SLAM implementations derive from:

- **EKF-SLAM:** historically, was the first implementation. In this method there is a state vector used to estimate robot's location and a set of point (that represents world's landmarks) with an error covariance matrix associated with the uncertainty of the predictions of those points. As the robot keeps moving, the state vector and the covariance matrix is updated using an EKF (Extended Kalman Filter). This is a widely popular method but suffer from computational costs.

- **Non-linear Sparse Optimization:** based in a graphical representation of the SLAM problem. In this method, the landmarks and the locations of the robot are seen as nodes in a graph, in which consecutive position nodes are connected, as well as the locations and associated landmarks. These connections represent soft constraints. Relaxing these constraints, we obtain the best estimation of the map and the best estimated motion path of the robot. This method is mostly used in offline SLAMs.

- **Particle Filter**: the estimated state is represented as a set of particles. One particle represents a single estimate of the real state. Thus, joining these estimations as a set of particles, the filters can estimate the posteriori-state distribution. It was proven that the particle filters make perfect estimation when the number of particle is close to infinite. One of the most popular particle-filter algorithm is the FastSLAM *(Montemerlo, et al. 2002)*.

2.4.2 Robot's Motion Planning and Navigation

Motion planning is used in robotic as the process of detailing a task into discrete motions. The research in robot motion planning can be traced back to the late 60's, during the early stages of the development of computer controlled robots. However, most of the effort is more recent and has been conducted since the 80's.

Motion Planning tries to address the *Piano Mover's Problem*, in which given a set of obstacles, the initial position and the final position, the *motion planner* has to find a path that moves the robot from the initial to the final position, avoiding the obstacles at all

times (see Figure 2.13). However, there are challenges in the real world that the moving robot has to address, such as physical laws (e.g., inertia, acceleration), uncertainty (e.g., location map, observation), geometric constraints (e.g., shape of a robot) or dynamic environment (e.g., a moving crowd).

Figure 2.13: A motion path generated for a mobile robot. The goal position and orientation is represented by the red arrow

Some properties of planning algorithms are:

- ***Completeness:*** determines if the algorithm finds a solution in finite time or is resolution/probabilistic complete.
- ***Optimality:*** determines if the algorithm finds the optimal solution(s)
- ***Complexity:*** determines what are the computational and memory demands
- ***Off-Line vs. On-Line:*** determines whether the algorithm need off-line pre-computing.
- ***Sensor-based:*** determines if the algorithm integrates a sensing step.

The earliest and most simple planning/navigation algorithm is the *Bug Algorithm*, which consists of purely sensor-based navigation biologically inspired. It assumes that the robot is able to detect obstacles and the direction to the goal. Its basic idea consists: following a straight line to the goal; if hitting an obstacle circle the obstacle clockwise; continue from the closest free point. Despite being quite simple, it proved to be non-viable for many cases, since it uses exhaustive search, it doesn't compute the optimal when obstacles are in between the robot and the goal, and it can be stuck in local minimum.

In Figure 2.14, we can observe the basic building blocks associated with navigation and motion planning in ground robots.

The global planner consists of a functional component in which, given a representation of the environment including known obstacles (e.g., a map), a current position and a goal position, it finds a free path from the current position to the goal position, i.e., a global plan. Nevertheless, it has to make some assumptions, such as a holonomic motion [4], a point representing the robot's position, a static and non-dynamic environment.

The local planner is generally in charge of obstacle avoidance and driving the robot to a local goal position. Thus, for the local planner the environment can change dynamically and not be entirely modeled. The local planner may modify or replan a globally generated plan and move the robot towards a goal without a global plan. It relies on information about the goal or sub-goal on the global path, local context information (localization) and recent sensor information (local map).

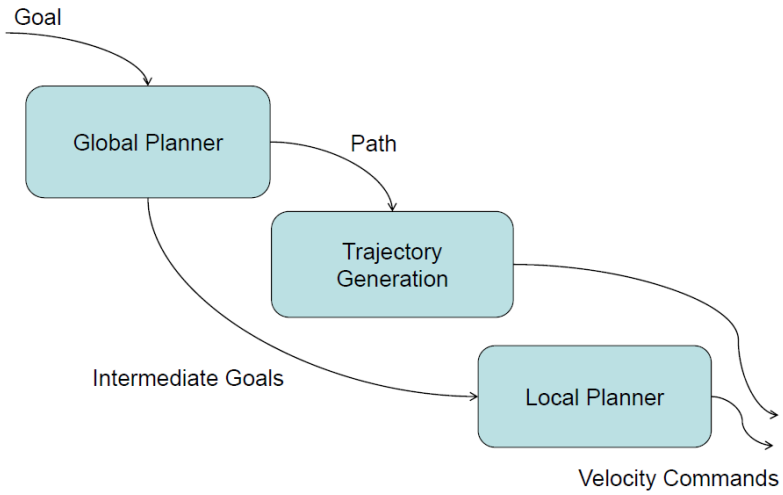


Figure 2.14: Building blocks for ground robot's Navigation and Motion Planning

There are many ways of representing the world in which a robot navigates. Below you are some of the most used types of representation:

- ***Grid-based Representations:*** represent the environment as a grid map. Grid cells are either free or occupied, though it may represent some uncertainty. These representations are very easy to use and easy to update, modify and combine. However, memory demand depends on the resolution and it proved not to be very efficient for sparse environments.

- ***Topological Representations:*** represent the environment as graphs or roadmaps. In the graph, the nodes represent specific configuration or spaces in the world,

[4] An holonomic motion allows a mobile vehicle/robot to immediately move in any direction without needing to turn first

while the edges represent the free connection between those configurations or spaces. The planning process consists in finding the shortest path in graph. However, it is overhead of creating the representation.

- ***Geometric Representations:*** represent the environment by its geometry, using geometric primitives like lines, splines, squares, polygons. It is a very compact and precise representation but it is difficult to construct using sensor data and large changes in the representation may occur due to small changes in the environment.

When using topological representations, the most popular algorithm to find paths in graphs is the A* Search (Hart, Nilsson and Raphael 1968). The A* Search is an extension of the Djikstra Algorithm, in which the estimated cost function for a node *n* consists on the sum of the cost from start so far with the estimated costs to goal from *n*. By using heuristics, it achieves better performance, with respect to time.

The LNP algorithm is used in local planning and consists in a method to find the minimum cost path based on linear programming which works well on grid maps. Summarily, given a path as points in a sample space, costs for traversing this space and a navigation function which represents the steepest gradient for any path starting in every given path's points, the algorithm returns a path with minimum costs.

Another algorithm used in local planning is the Potential Field which models the robot as charged particle in a potential field. Then goal has an attractive force while the obstacles have a repulse fore. The gradient of the potential field provides a force and the robot rolls down a surface towards the goal. It is an algorithm easy to implement but it is hard to find proper parameters and can make the robot trapped in a local minima.

In 1997, Dieter Fox, Wolfram Burgard, and Sebastian Thrun developed The Dynamic Window Approach (Fox, Burgard and Thrun 1997). Unlike other avoidance methods for local planning, the dynamic window approach is derived directly from the dynamics of the robot, and is especially designed to deal with the constraints imposed by limited velocities and accelerations of the robot. This approach assumes a differential drive robot that moves along arcs. It uses a dynamic window of reachable velocities in the next cycle, given by the dynamic of the robot, considering only admissible velocities yielding a trajectory on which the robot is able to stop safely. The objective function includes a measure of progress towards a goal location, the forward velocity of the robot and the distance to the next obstacle according to the trajectory.

To conclude this sub-chapter, it is important to mention that neither global nor local methods are enough while building an efficient navigation system for a mobile robot. Global methods fail to address local variations, uncertainty and environment's dynamics, while local methods get trapped in local minima. Therefore, the best solution consists in combine these both methods. While building a robot control architecture with a navigation system, it is important to discriminate the different navigation building blocks according to their working frequency. Generally, good approaches prioritize actuator control and obstacle avoidance (local planners) enhancing their control loop frequencies, while path planning and global planning's control loop has lower frequencies.

2.5 Software Platforms for Robotic Controllers

In order to make a robot to successfully execute tasks, proper robot software has to be implemented. This software usually consists of instructions that control robot's actions and provide information regarding required tasks. Nevertheless, programming robots can be a complex and challenging process, especially if the robot is composed of several hardware units which required specialized processes and synchronous inter communication. When developing robot software, many researches are more interested in higher levels of abstraction, i.e., the implementation of the robot control architecture that manages the robot's mission. However, in order to implement a full robot application, one must deal with several levels of abstraction, i.e., must handle with different programming languages and computer architecture levels and must take into account the low-level controllers for the robot's hardware units. For this reason, **Software Platforms for Robotic Controllers** have been developed in order to facilitate the software integration in a robot, to deal more easily with the complexity of the robot software and to define common use software (standard API) that control several robot's units. A software platform can be seen as a middleware between our application and the robot's operating system of drivers that control robot's hardware (see Figure 2.15).

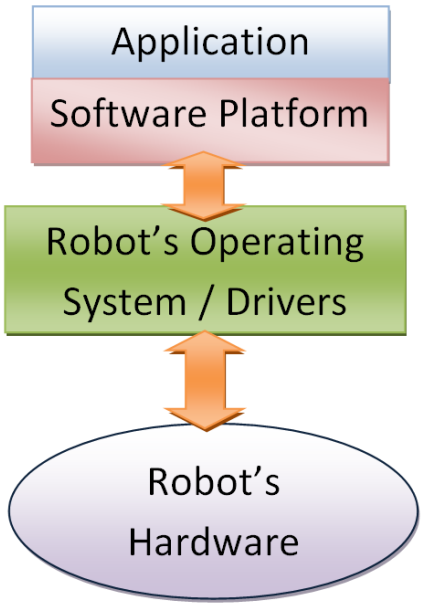


Figure 2.15: Generic Software Platform for Robot Controllers

Nowadays, we can find many software platforms, frameworks libraries and operating systems for many kinds of robots and robot's missions that can support several programming languages (see Figure 2.16). In the following sections, we are going to describe three well-known robot's software platforms and frameworks which are currently very used in robotic research: Urbi, OpenRDK and Orca. Another very popular robotic framework is ROS (Robot Operating System), which was used for the work described in this document. This framework is going to be detailed in the section 4.2 Software Platform – ROS.

Framework	Concurrency model	Information sharing	Tools	Focus
OROCOS[1] [3]	call-backs, threads	lock-free data ports (CORBA)	remote inspection, logging	low-level devices
Orca[2] [8]	processes	ICE[13]	remote inspection, logging	mobile robots
CARMEN[3]	processes	IPC	logging, visualization	mapping and navigation
OpenRTM-aist[4] [1]	threads	CORBA	configuration GUI	general robotics
Microsoft Robotics Studio[5]	processes	HTTP/DSSP via DSS	3D simulator	general robotics
Player[6] [4]	threads (server)	client/server, proprietary over TCP	2D and 3D simulators	low-level device drivers
MOOS[7]	processes	centralized, proprietary over TCP	logging, viewers	mobile robots
CLARAty[8] [9]	threads, processes	relies on ACE[14]	none	real-world systems
MARIE[9] [5]	processes	many (3rd party)	configuration GUI	connecting different frameworks
MOAST[10]	processes	NML/RCS[11]	logging, visualization	USARSim, mobile robots
MIRO[11] [10]	processes	CORBA	logging	mobile robots
SPQR-RDK [6]	call-backs, threads	proprietary over TCP	remote inspection	mobile robots
OpenRDK	threads	shared memory, proprietary TCP/UDP	remote inspection, logging	mobile robots

[1] http://www.orocos.org
[2] http://orca-robotics.sourceforge.net
[3] http://carmen.sf.net
[4] http://www.is.aist.go.jp/rt/OpenRTM-aist
[5] http://msdn2.microsoft.com/en-us/robotics/
[6] http://playerstage.sf.net
[7] http://www.robots.ox.ac.uk/~pnewman/TheMOOS/
[8] http://claraty.jpl.nasa.gov/
[9] http://marie.sf.net
[10] http://moast.sf.net
[11] http://www.isd.mel.nist.gov/projects/rcslib
[12] http://smart.informatik.uni-ulm.de/MIRO/
[13] http://zeroc.com/ice.html
[14] http://www.cs.wustl.edu/~schmidt/ACE.html

Figure 2.16: Some of existing software robotic framework. Retrieved from (Calisi, et al. 2008)

2.5.1 Urbi

Urbi is an open source software platform to control robots or other complex systems. Its main goal is to help making robots compatible and simplify the process of writing programs and behaviors for these robots. Thus, it provides all the needed features to control the execution of various components such as actuators, sensors and software devices that provide features like face recognition.

Urbi relies on a middleware architecture that coordinates components named *UObjects*. *UObjects* are written in C++, with a provided library that can describe robot's motors, sensors and algorithms. They can run on top of Gostai Runtime. Components with the *UObjects* are supported by the *urbiscript* programming language, which is a dynamic, prototype-based, object-oriented scripting language. *Urbiscript* glues the components together to describe high level behaviors with embedded parallel and event-driven semantic. The Urbi middleware architecture makes possible to use remote components as if they were local, to allow concurrent execution and to make synchronous or asynchronous requests. For the general Urbi architecture, see Figure 2.17.

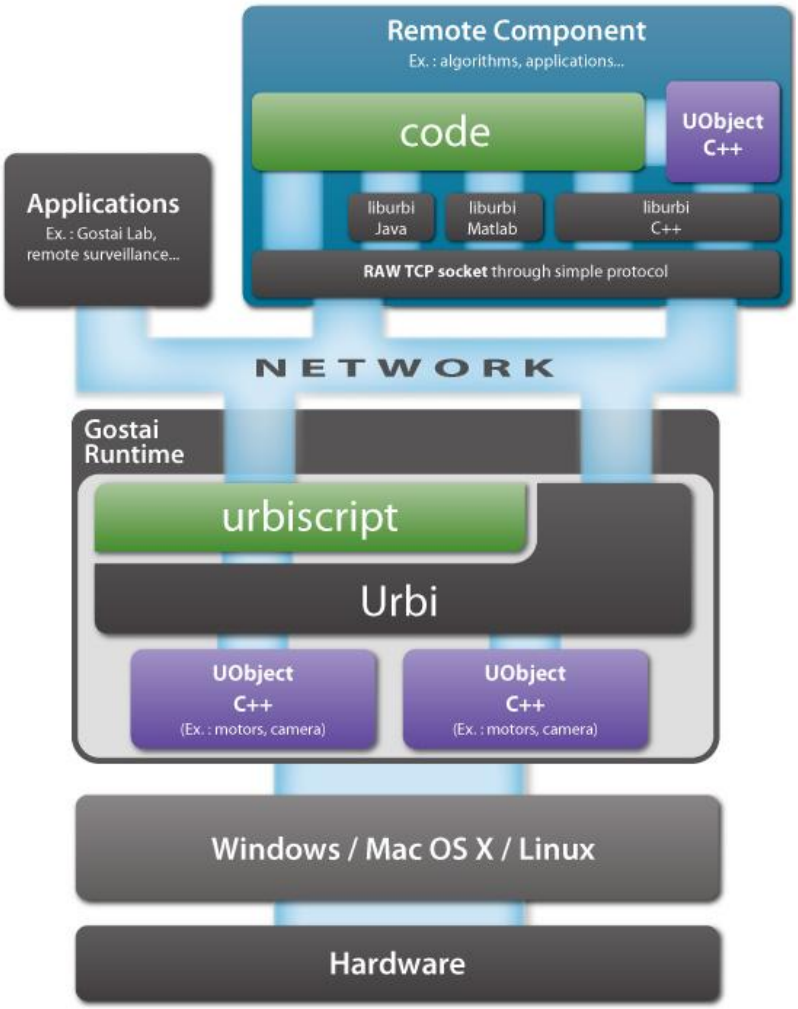


Figure 2.17: Urbi's Integration into a project

Urbi is compatible with famous robots such as the Sony's *Aibo*, the humanoid *Nao*, from *Aldebaran Robotics* and *Pioneers*. It is also compatible with a few robotic simulators, such as *Cyberbotics's Webots*. For more information regarding the Urbi software platform, see (GOSTAI 2010).

2.5.2 OpenRDK

OpenRDK is modular framework for robotic software, designed and implemented by D. Calisi and A. Censi, from the SIED Laboratory and RoCoCo Laboratory of the University of Rome, Italy (Calisi, et al. 2008). OpenRDK is written in C++ and runs on Unix-like operating systems (such as Linux, OS X). It is focused on rapid development of distributed robotic systems, and has been designed following users' advice. By now, OpenRDK has been successfully applied in diverse applications with heterogeneous, being very accessible for the use and improvement by many research groups, since it is released as open source. It is currently being used in the RoCoCo Laboratory (referred

above) and in the Intelligent Control Group of the Technical University of Madrid, Spain.

In OpenRDK, the main software process is called agent, which can contain several modules (see Figure 2.18). A module is a single thread inside the agent's process and can be loaded and started automatically one the agent process is running. Modules communicate using blackboard-type object called repository in which they publish some of their internal variables called properties. A module can access its own properties and other module's properties, within the same agent or remotely, through a global URL-like addressing scheme. OpenRDK provides built-ins for concurrency management need to control the access of the shared memory. All these entities are implemented in the OpenRDK core, thus all a developer is requested to do is to create a new module which is a relative "easy" task. This way, the developer can concentrate on the real problem, without having to care much about the framework.

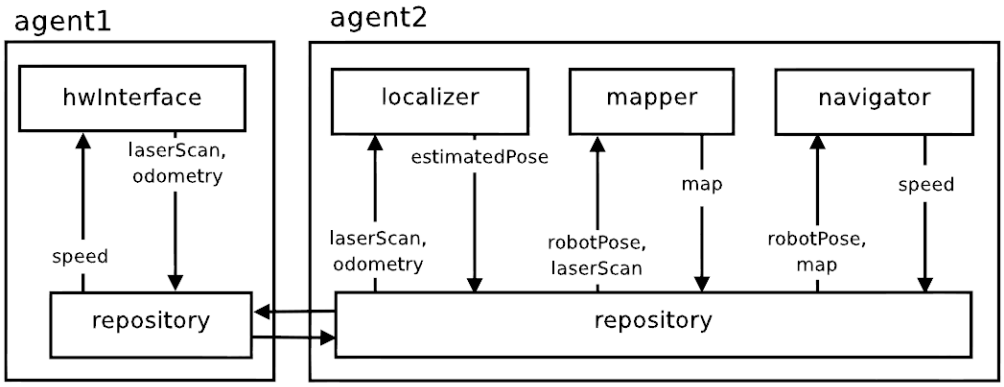


Figure 2.18: Agents and module interconnection in OpenRDK (Calisi, et al. 2008)

2.5.3 Orca

Orca is an open-source framework for developing component-based robotic systems (Makarenko, Brooks and Kaupp 2006). It began as a part of the EU-funded OROCOS Project that aimed to develop an Open-Source Robotic Control System, with the collaboration of universities of Sweden, Belgium, France and Germany. This framework provides means for defining and developing building-blocks (components) which can be pieced together to form arbitrarily complex robotic systems, from single vehicles to distributed sensor networks. For the developers, the project's main goal is to promote software reuse in robotics. Currently, Orca is used in several projects and robotics research centers, such as the Center for Collaborative Control of Unmanned Vehicles of the University of California, and Cornell University Autonomous Systems Lab.

Orca adopts a Component-Based Software Engineering (CBSE) approach without applying any additional constraints. This offers developers the opportunity to source existing plug-in software components, rather than building everything from scratch. It also uses a commercial open-source library for communication and interface definition

and provides tools to simplify component development, making them strictly optional to maintain full access to the underlying communication engine and services. Additionally, Orca uses a cross-platform development tools. This makes Orca a general, flexible and extensible framework.

Orca enables inter-component communication using the Ice middleware, which enables the Orca components to be implemented in different programming languages running on different operating systems (see Figure 2.19). The Ice core library manages all the communication tasks using a protocol which includes optional compression and support for both TCP and UDP.

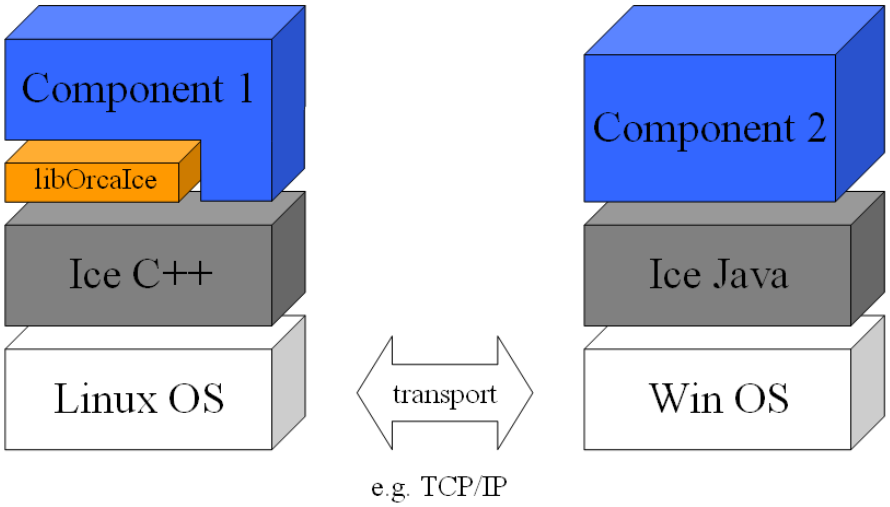


Figure 2.19: Orca components and the Ice Middleware (Makarenko, Brooks and Kaupp 2006)

3 ASys and The Operative Mind

In this section, you can find a summary description of the research project associated with this master project: the ASys project, as well as the architectural framework in which the control architecture described in this work is based: The Operative Mind.

3.1 ASys Project

The ASys Project is a long term project of the Autonomous Systems Laboratory research group of the Technical University of Madrid [5]. Its main focus is the development of technology for the construction of autonomous systems. ASys tries to fill the current necessities of building complex (many times distributed) control systems that deal with higher degrees of uncertainty, providing robust autonomy at the required level. What makes ASys different from other projects in this field is the extremely ambitious objective of addressing *all the domain of autonomy*. This is captured by its motto: *engineering any-x autonomous systems*.

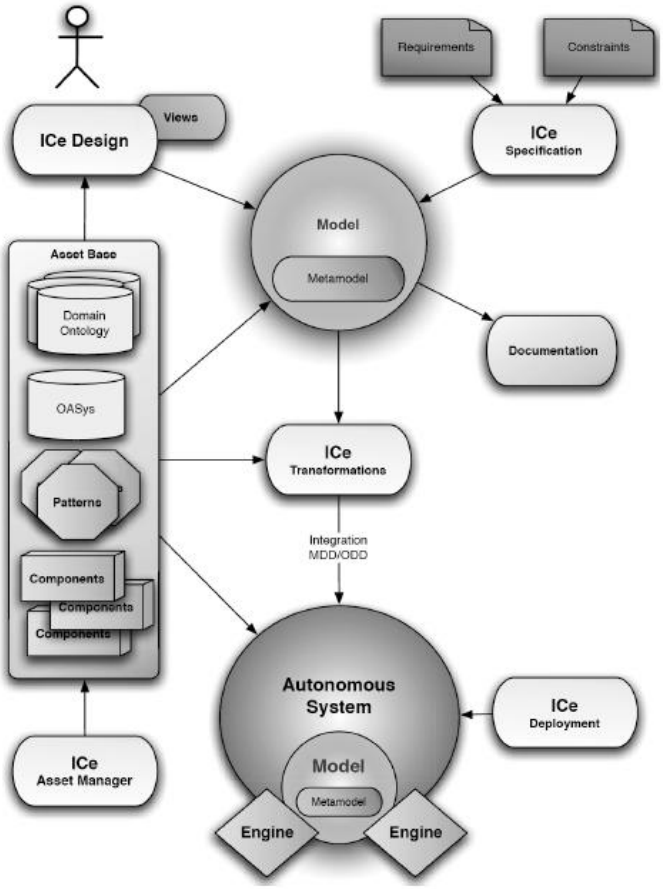


Figure 3.1: Elements of the ASys Research Program

[5] http://www.aslab.org/public/

ASys project considers different elements to materialize its objectives: an architecture-centric design approach, a methodology to engineer autonomous systems based on models, and an asset base of modular elements to fill in the roles specified in the architectural patterns. Some elements of the ASys project can be seen in Figure 3.1.

ASys specifies an engineering process that covers from the initial specification and knowledge, to the final product, i.e., the autonomous system. The first stage of the research program focuses on ontologies, as a common conceptualization to describe domain knowledge. Both a survey of existing domain ontologies and the development of an ontology for the domain of autonomous systems are addressed. As a result, the OASys ontology was specified (Bermejo-Alonso 2010).

A cornerstone of the ASys program is the use of design patterns, as the core vehicle for reusable architecture exploitation (see 3.3 - ASys Cognitive Patterns). Generally, design patters present solutions to recurring design problems in a certain context.

One of the main pillars in ASys is the model-based approach: a truly autonomous system will be continuously using models to perform its activity. Additionally, ASys can exploit models of itself to drive its operation or behavior. In this way, model based engineering and model-based behavior merges into a single phenomenon: *model-based autonomy*. The type of models and use of models to be specifically developed for the autonomous system are initially considered. User and designer requirements, and constraints imposed by the system itself will guide the development of the models. The ASys Model Development Methodologies will address this model characterization and development. The next stage is to extract from the built models a particular view of interest for autonomous system. Unified functional and structural views are considered critical, as they provide knowledge about the intentions and the behaviors of the autonomous system.

3.2 ASys Cognitive Principles

As previously commented, one of the ASys main pillars is the model based approach to cognition. From a control engineering perspective, ASys research program considers cognition as the exploitation of knowledge –i.e. models- to realize control. Models are understood as the part responsible for maintaining behavior directed towards systems objectives while satisfying certain conditions. In biological systems we have minds embodied in brains whereas in artificial systems we have control laws and control architectures nowadays mainly embodied in computers. Departing from the basic principle: *a system is said to be cognitive if it exploits models of other systems in their interaction with them*, ASys research works have started building up the ASys principle approach to cognition and consciousness (Sanz, Lopez, et al. 2007). As a result, eight principles of the model-based approach of cognition in ASys were formulated:

1. **Model-based cognition:** a cognitive system exploits models of other systems in their interaction with them.

2. **Model isomorphism:** an embodied, situated, cognitive system is as good as its internalized models are.

3. **Anticipatory behavior:** except in degenerate cases, maximal timely performance is achieved using predictive models.

4. **Unified cognitive action generation:** generating action based on an unified models of task, environment and self is the way for performance maximization.

5. **Model-driven perception:** perception is the continuous update of the integrated models used by the agent in a model-based cognitive control architecture by means of real-time sensory information.

6. **System awareness:** a system is aware if it is continuously perceiving and generating meaning from the continuously updated models.

7. **System Self-awareness/consciousness:** a system is conscious if it is continuously generating meaning from continuously updated self-models.

8. **System attention:** attention mechanisms allocate both physical and cognitive resources for system perceptive and modeling processes so as to maximize performance.

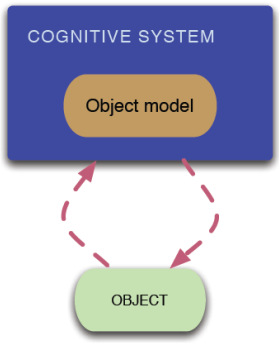


Figure 3.2: Models as cognitive relations of a system with an object (Sanz, Lopez, et al. 2007)

One of the most relevant key aspects is the integration of the model and the metamodel (related with Principle 7, see above). One big difference between being aware and being conscious comes from the capability of action attribution to the system itself thanks to the capability of making a distinction between self and the rest of the world. This implies that conscious agents can effectively understand – determine the meaning- the effect of its own actions (computing the differential value derived from self-generated actions, i.e., how its own actions change the future).

3.3 ASys Cognitive Patterns

A cornerstone of the ASys program is the use of design patterns as the core vehicle for reusable architecture exploitation (Sanz and Zalewski 2003). Design patterns present solutions to recurring design problems in a certain context. ASys patterns could be classified in two categories:

- **Architectural patterns:** express the structural organization of an autonomous system, i.e., they realize the architecture.

- **Domain patterns:** describe a mechanism to solve a concrete but recurring problem in a particular context.

Patterns are used in ASys independently from the OASys ontology. Domain patterns describe interactions of the system's components and with the environment, by using the conceptualization of the ontology, that represent design solutions so that the behavior of the system fulfills the engineering requirements. They model the intended – designed – system's dynamics with its environment. On the other hand, architectural patterns do the same for the internal system's organization in between the ontological elements that conceptualize the system itself. Thus, all system patterns will not only be specified departing from the OASys concepts, but eventually will become part of the ontology itself, modeling the relations and interactions between them as designed by engineers.

One of the accomplished works within the ASys Research Program is that described in (Lopez 2007), which defined a general framework for autonomous systems. This work proposed an unified theory of perception, which can be also considered as a Cognitive Pattern, in which the specified entity "Perceptor" corresponds to a global entity which tries to generalize the processes of perception that are present in any cognitive architectures.

The cognitive pattern more relevant for this work is the one reported in (Hernandez, Lopez and Sanz 2009) as the *epistemic control loop* (see Figure 3.3). This article defines a cognitive control architecture, named *Operative Mind* (*OM*) and inspired, among others, by Albus's RCS. This control architecture consists of a network of "cognitive nodes", in which each of them is realizing a control loop following the pattern of the epistemic control loop and not necessarily organized into a hierarchy, but connected as required by the current task and global state, i.e., system state and environment state as perceived by the system. These nodes can have different spatiotemporal resolutions and span several levels of abstraction. According to the ASys cognitive approach, each node in the architecture maintains and exploits models of the world and the system physicality, to the extent relevant for its operation.

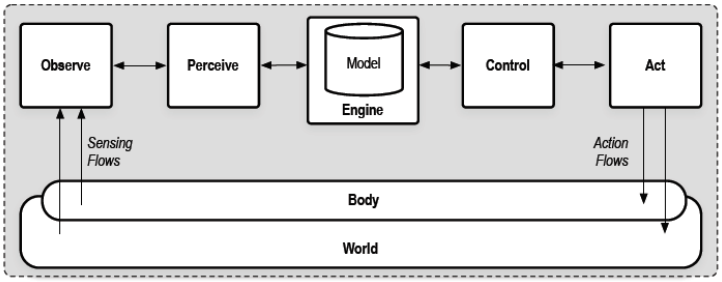


Figure 3.3: The core epistemic control loop in The Operative Mind (Hernandez, Lopez and Sanz 2009)

In the OM architecture, there are also *meta-nodes* that monitor and control the operation of nodes, and are also patterned after the epistemic control loop 2 (see Figure 3.4). More interestingly, meta-nodes are also explicitly modeled, so they can operate upon themselves, closing the controlling-the-controller regression loop.

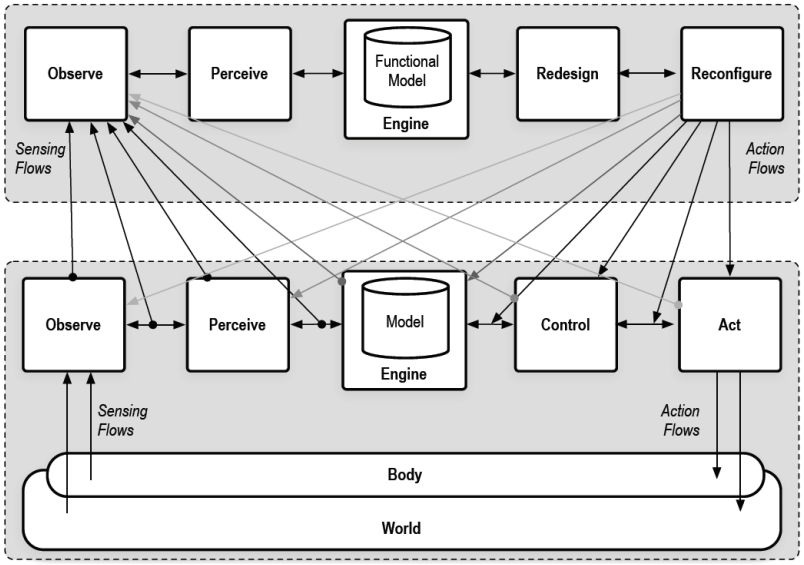


Figure 3.4: The epistemic control loop applied in meta-nodes that monitor and control other nodes (Hernandez, Lopez and Sanz 2009)

3.4 Beyond Current State of the Art

As mentioned in 2.1 - Agent Architectures, there are several agent architecture approaches, each with different characteristics and trade-offs. In most cases, these architectures tend to be poorly flexible, difficult to expand or designed to solve a close set of problems. With reactive architectures, the agent's behaviors tend to be predictable, homogeneous and non-viable for complex problems or dynamic environments. Conventional deliberative or hybrid architectures deal with the problem of complexity generally by adding components in charge of reasoning, decision making over a built a world model or merging it with reactive processes. Nevertheless, these

architectures proved to be rigid as well as very difficult to implement in practical problems and have only been successfully implemented in very closed-domains and under controlled environments.

For higher levels of intelligence, autonomy and robustness one has to attend for less conventional architectures, such as the cognitive architectures (see 2.2 - Cognitive Architectures). The researchers that work with these architectures believe that we can improve the overall intelligence (adaptability, autonomy, robustness, etc.) of artificial systems by understanding and replicating the intelligence found in biological entities or processes, such as the human mind. In a way, this approach seems to be highly appealing because many of the problems that machines are not capable of solving can be easily done by humans (such as autonomous driving). However, cognitive architectures tend to be extremely complex, and most of them are still in developments. The main problem in developing these architectures is the lack of a formal definition of the theory of the human mind, which explains the cognitive processes that occur in the human brain that are able to make us intelligent.

The cognitive control architecture *Operative Mind* described in (Hernandez, Lopez and Sanz 2009) is a cognitive architecture in the way that it tries to replicate some of the cognitive phenomena that occurs in the human mind, such as introspection, self-awareness and even consciousness. This is possible thanks to the existence of meta-nodes that monitor and control the operation of the architecture's functional nodes. This meta-control layer adds the flexibility and the reconfiguration capabilities that lacks in conventional architectures. Compared to other cognitive architecture, the *Operative Mind* is proved to be less restrictive and more adaptable, since it can be built on top of the majority of control architecture, while they are organized in functional nodes that follow the core epistemic loop (see Figure 3.3). Thus, unlike RCS in which the control architecture have to be organized into a hierarchy, the *Operative Mind* can be adapted from a conventional agent architecture, in which a meta-control layer is added to enhance its adaptability and robustness by means of self-introspection and self-reconfiguration.

The *Operative Mind* can also be adapted to a fault-tolerant system in which the meta-control layer detects a fault and properly reconfigures the control architecture for the fault accommodation. The advantage of the *Operative Mind* over systems described in (*2.3 Fault-tolerance in Autonomous Systems*) is that the manipulation of the control system by the meta-controller can be made, not only in cases of fault, but in other special occasions in which a reconfiguration of the system enhances the efficiency of a mission. Thus, unlike other conventional fault-tolerant systems, reconfiguration of the system is not exclusively fault-reactive, but can be triggered by autonomous model's introspection.

4 Hardware and Software Platform

This chapter presents the hardware and software available to implement the developed control system, that would be used in the surveillance navigation mission, described in the next chapter.

4.1 Hardware Platform - Higgs

Higgs robot is part of the ASys Robot Control Testbed (RCT) and has been selected as the target demonstration of the cognitive robotic control architectures fruit from the ASys and ASLab research work.

Figure 4.1: RCT Robotic Platform - Higgs

Higgs is a mobile robotic system that consists of a base platform and different interconnected subsystems to cover a wide range of capabilities. The research aim is to provide the mobile robot with the necessary cognitive capabilities and an intelligent control system, as to perform complex tasks. In the following sections, we are going to describe the hardware components of the robot (some of which can be appreciated in Figure 4.1), and the software platform (or middleware) that is used to control the hardware systems of the robot.

4.1.1 Base Platform (Pioneer 2-AT8)

The base platform of Higgs is a mobile robot Pioneer 2-AT8, which has been designed by ActivMedia Robotics (see Figure 4.2). It is a robust platform that includes all the necessary elements to implement a control and navigating system, specially designed for outdoor applications. Additional systems and elements can be attached to this platform. The base platform is given the ASLab name Higgs, as a reference to the Higgs's boson.

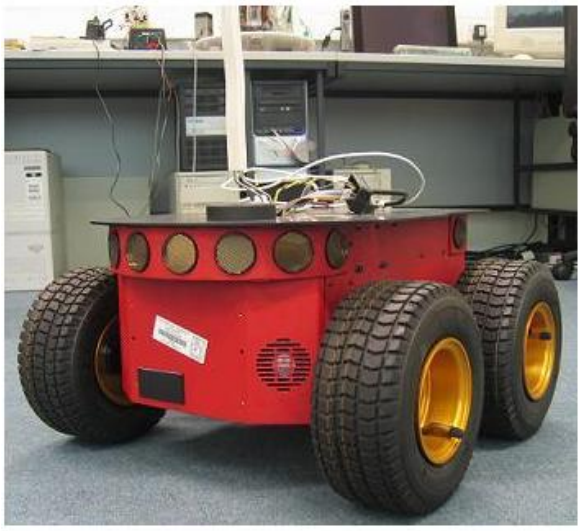


Figure 4.2: The Pioneer 2-AT8 platform used to implement the RCT Higgs Robot

This base platform is a small size mobile robot, with a support structure made of aluminum. Its total weight is of 15 kg, being capable of carrying up to 40 kg. From a hardware viewpoint, the mobile robot consists of different elements:

- **Robot Panel:** it is the superior platform of the robot, designed for a later assembly of new elements such as cameras or laser systems.

- **Robot Body:** it is a box-shaped element made of aluminum. It contains the batteries, the actuators, the electronic circuits and the rest of elements. It also allows attaching additional elements such as an onboard PC, more modern or additional sensors.

- **Control Panel:** it is the access panel to the robot's microcontroller, planed in the robot panel. It consists of several control buttons, robot status leds (robot switch on, microcontroller status, battery charge) and a serial port RS-232 to be used as an input and output communication link with an external PC.

- **Sensors:** the mobile robot is provided with two arrays of eight sensors each, which allows the detection and location of objects in the mobile robot environment. The arrays are placed at the front and at the rear part of the robot.

- **Actuators:** the robot contains 4 Pittman motors GM9236E204. Each one includes an optical encoder to determine the robot's speed and position.

- **Microcontroller:** it is a Hitachi H85 microcontroller consisting of different elements (memories, serial ports, inputs, outputs, 8 bit bus) which carries out different operations such as trigger and registers the sensors' signal, controls the actuators, and some other low-level operations.

- **Bumpers:** they are additional elements attached to the platform, 5 at the front and 5 at the rear.

- **Power:** there are three batteries 12 VDC 7 Ah-h, located at the rear part of the robot. They provide 252 W-h, which assure several hours of autonomy movement to the robot. Their status can be checked in the corresponding led of the control panel.

In addition to the hardware, the mobile robot has different software elements provided by the manufacture:

- **AROS (ActiMedia Robotics Operating System):** it is the operating system, consisting in server processes running on the Hitachi microcontroller in Higgs. It is a low–level software in charge of regulating the motors' speed, sonars' signal, encoders' signal and other low-level tasks. This software will also communicate the obtained information to other client software applications through the RS-232 serial interface.

- **ARIA:** it is an applications-programming interface (API) based on C++ to control the robot. It acts as the client in the client–server topology. It allows to program high–level software applications, such as intelligent behavior (obstacle avoidance, object recognition, wandering, exploration, etc). The robot control is based on direct commands, movement commands or abstract–level actions.

4.1.2 Onboard Systems

On the base platform, different devices have been attached to expand the original range of functionalities of the mobile robot (see Figure 4.1):

- **Laptop:** it is a lightweight high functional laptop, located in the back part of Higgs's top. Its model is Sony Vaio TX2HP. As information links, it has two USB 2.0, WIFI IEEE 802.11b/g, one Ethernet and a Bluetooth 2.0. Its autonomy is of approximately 2 hours. It also has a Intel Pentium M 1.1 GHz as a processor and a memory of 512 MB DDR2. The operating system running on it is the Ubuntu 10.04.3 LTS (Lucid Lynx).

- **Laser:** it is a laser scanner for mobile robotics applications, placed at the front part of the robot panel. It is screwed to the front part of Higgs's top and includes a mechanism for tilting by means of a servo. The model of the laser is Sick

LMS-200. All of the sensors operate by shining a laser off of a rotating mirror. As the mirror spins, the laser scans 180°, effectively creating a fan of laser light. In ideal conditions, the LMS-200 is capable of measuring out to 80m over its 180° arc. For an object with only 10% reflectivity (such as matt black cardboard), the LMS 200 can measure out to 10m. The sensor is best for indoor use as it can be dazzled by sunlight (causing it to give erroneous readings).

- **Kinect:** it is a motion sensing input device owned by *Microsoft* designed for the Xbox 360 video game console. The Kinect work as a 3D scanner, using a technology from *PrimeSense* which interprets 3D scene information from continuously-projected infrared structured light. Thus, it is able to capture video data in 3D under any ambient light conditions. The sensor has an angular field of view of 57º horizontally and 43º vertically. It also has a motorized pivot that is capable of tilting the sensor up to 27º either up or down, and a four microphone array which enables the sensor to conduct acoustic source localization and ambient noise suppression.

- **Arduino board:** Arduino consists of an open-source hardware platform based in a board with a microcontroller, designed to make the process of using electronics in multidisciplinary projects more accessible. The Higgs's onboard Arduino is used to control de following devices:

    - Compass
    - Accelerometers
    - Battery sensors
    - Laser heading
    - Power board

  These devices are connected to the Arduino Mega commercial board through a custom made extension board. We interact with the Arduino using a CORBA module that has been written in the JAVA language whilst the test client in C++. The embedded program inside the Arduino has been developed using the official IDE based on the processing IDE and the libraries associated to it. Both the embedded program inside the Arduino and the Java servant communicate through a USB connection with the serial profile. The Arduino starts the communication protocol by sending all the parameters and sensor readings to the servant, and then the servant optionally answers with the order.

- **Power board:** consists simple power board for controlling power to other devices. It is a attached to the front of the laptop support. It was specifically designed for the need of the investigator at ASLab to rest algorithms with partial malfunction of a robot. The power board is in control of the power of up to 9 devices in the robot (such as the Kinect and the laser device), through channels, that can be manually shot down by means of a physical switch.

4.1.3 Supporting Systems

Along with the base platform and the onboard systems, Higgs includes some additional supporting systems to complete its functioning:

- **Wireless Network**: it is an additional subsystem included in the testbed to allow the communication with the wireless local network in the ASLab laboratory. It consists of a wireless card (locates inside the onboard laptop) and an access point (connected to the laboratory LAN).

- **Wii Control**: It is the primary controller for Nintendo's Wii console. In the context of ASLab, it is used as a portable device to remote control the robot. A main feature of the Wii Remote is its motion sensing capability that is enabled due to the use of an accelerometer and optical sensor technology.

4.2 Software Platform – ROS

In this section, the ROS middleware robotic platform is going to be partially explained. As all software implementations related to this work were made over ROS, it is important catch the main aspects of this platform for a better understanding of the next chapters.

4.2.1 Introduction

ROS stands for Robot Operating System, and it consists of an open-source, meta-operating system for robots. It is similar in some aspects to other robot frameworks, such as Orca, Urbi and OpenRDK (see 2.5 - Software Platforms for Robotic Controllers). ROS provides the services you would expect from an operating system, including hardware abstraction, low-level device control, implementation of commonly-used functionality, message-passing between processes, and package management. It also provides tools and libraries for obtaining, building, writing, and running code across multiple computers (Conley 2011).

ROS is specially designed for complex mobile manipulation platforms with actuated sensing, and compared to other robot software platforms it makes it easier to take advantage of a distributing computing environment.

Some of its main characteristics, that can partially justify our selection of ROS as our main robot platform, are:

- Based on a graph architecture: processing takes place in nodes that may receive, post and multiplex sensor, control, state, planning, actuator and other messages.

- It doesn't wrap the main(), thus it can be written for be used with other robot software frameworks.

- Language independence: it can be easily implemented in any modern programming language, such as Python, C++ and Lisp.

- Easy testing: ROS provides built in integration for testing and monitoring that make it easier to handle complex systems.

- Scaling: ROS is appropriate for large runtime systems and for large development processes.

Currently, ROS runs on Unix-based platforms, being primarily tested on Ubuntu and Mac systems, though having support for more distribution. While a port to Microsoft Windows for ROS is possible, it has not yet been fully explored.

4.2.2 ROS Concepts

At the filesystem level, ROS is organized in **packages**, which are the main unit of organizing software. A package may contain ROS Nodes (runtime processes), a ROS-dependent library, datasets, configuration files, or anything else that is usefully organized together. Inside a package we can find a **Manifest** (which provides metadata about a package, such as dependencies), **messages** and **service** types. These latter consist of message and service descriptions which define the data structures for messages and services exchanged in ROS, respectively. In ROS, **stacks** refer to a collection of packages that provide aggregate functionality.

At the ROS computation graph level, we have **Nodes** which are the processes that perform computation. ROS Nodes can interact trough other nodes in two ways:

- Through ROS **topics**: Messages are routed via a transport system with publish/subscribe semantics. A node sends out a message by publishing it to a given topic. A node that is interested in a certain kind of data will subscribe to the appropriate topic

- Through ROS **services**: The publish/subscribe model is a very flexible communication paradigm, but its many-to-many, one-way transport is not appropriate for request/reply interactions, which are often required in a distributed system. Request/reply is done via services, which are defined by a pair of message structures: one for the request and one for the reply. A providing node offers a service under a name and a client uses the service by sending the request message and awaiting the reply.

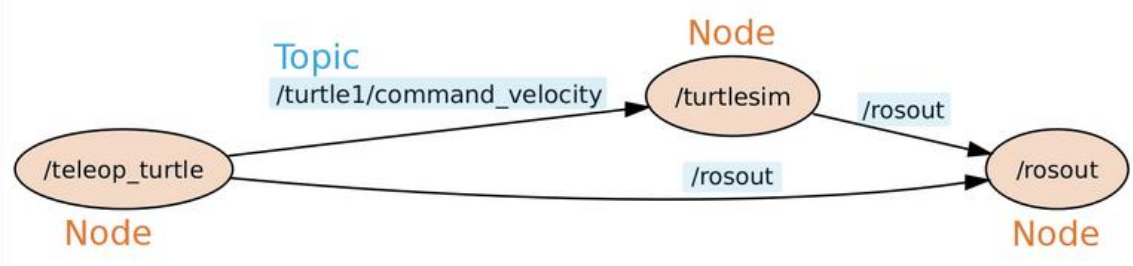


Figure 4.3: ROS nodes and topics (Wise 2011)

The ROS **Master** provides name registration and lookup to the rest of the Computation Graph. Without the Master, nodes would not be able to find each other, exchange messages, or invoke services. Nodes communicate with the Master to report their registration information. As these nodes communicate with the Master, they can receive information about other registered nodes and make connections as appropriate. The Master will also make callbacks to these nodes when the registration information is changed, which allows nodes to dynamically create connections as new nodes are run.

There is also a **Parameter Server** that allows data to be stored by key in a central location, being currently part of the Master. These parameters managed by the Parameter Server are often associated with some nodes. Thus, one can access and change nodes' parameters transparently and dynamically through this server.

4.2.3 ROS Client libraries and Nodelets

ROS client libraries are a collection of code that facilitates the task of programming in ROS. This is made through the accessibility via code of some of the main ROS concepts. In general, these libraries enables you write ROS nodes, publish, subscribe to topics, write and call services and use the Parameter Server (see 4.2.2 - ROS Concepts).

The main client libraries, as stated by (Conley 2011), are:

- ***roscpp****: roscpp is a C++ client library for ROS. It is the most widely used ROS client library and is designed to be the high performance library for ROS.*
- ***rospy****: rospy is the pure Python client library for ROS and is designed to provide the advantages of an object-oriented scripting language to ROS. The design of rospy favors implementation speed (i.e. developer time) over runtime performance so that algorithms can be quickly prototyped and tested within ROS. It is also ideal for non-critical-path code, such as configuration and initialization code. Many of the ROS tools are written in rospy to take advantage of the type introspection capabilities. The ROS Master, roslaunch, and other ros tools are developed in rospy, so Python is a core dependency of ROS.*
- ***roslisp****: roslisp is a client library for LISP and is currently being used for the development of planning libraries. It supports both standalone node creation and interactive use in a running ROS system.*

There are also some experimental client libraries, such as **rosjava** and **roslua**, which will be available soon.

As for the ROS nodelet, they are designed to provide a way to run multiple algorithms on a single machine, in a single process, without incurring copy costs when passing messages between processes. *roscpp* has optimizations to do zero copy pointer passing between publish and subscribe calls within the same ROS node. To do this, nodelets allow dynamic loading of classes into the same node, however they provide simple separate namespaces such that the nodelet acts like a separate node, despite being in the same process. There is a ROS package available in the ROS repositories, which provides both the nodelet base class needed for implementing a nodelet, as well as the *nodeletLoader* class used for instantiating nodelets. They authors of this package are Tully Foote and Radu Bogdan Rusu.

4.2.4 ROS Repositories

One of the main features and advantages of ROS is the large set of repositories available for any open-source robot programmer. ROS has created a community of robot software developers, making available a large set of packages and stacks, distributed in several repositories. Some stacks available are specially designed for some specific robots, but through some analysis one can fit them into another robot, if hardware integration is proved possible. The access to ROS's packages, stacks and repositories can be made through this link: http://www.ros.org/browse/list.php. Some stacks provide a complete documentation, describing the contents and the proceedings for integrations as well as some tutorials. One of the main ROS goals is push forward the robotic developments through software reuse, thus, the importance of these repositories is significant.

Some of the main stacks include utilities the following:

- Hardware integration of several robotic sensors, actuators, Arduino boards and several robot hardware platforms;

- 2D Navigation, Autonomous Localization, 2D Path Planning System and SLAM implementations;

- Computer Vision techniques for several appliances: Visual SLAM, Face Tracking, Object Recognition;

5 Higgs's Mission

Higgs[6]'s mission, described in this chapter, has been selected according to the available hardware and software platform, already mentioned in the previous section 5. It was defined with the intention to build a reconfigurable control system, by capturing the functional requirements of the control architecture and the control system to be implemented in Higgs. This mission consists in a fully autonomous surveillance of the laboratory room *Sala de Calculo* located in the DISAM (Department of Automation, Electronic and Computer Engineering) of the Technical University of Madrid (UPM).

The *Sala de Calculo* consists of a small laboratory room, with tables at the center and at the sides of the room, which form a rectangular corridor (see Figure 5.1).

(a) (b)

Figure 5.1: *Sala de Calculo*, DISAM, Technical University of Madrid, Spain

Higgs surveillance mission involves navigating autonomously through the main rectangular corridor of the *Sala de Calculo*, in a continuous path, making a complete(s) lap(s), either in clockwise or counter clockwise direction. In Figure 5.2 you can see a valid surveillance path (blue dotted line), Higgs's initial position (green circle) and some of the path's waypoints (red circles).

[6] Higgs as the Higgs robot, part of the ASys Robot Control Testbed of the ASLAB (see 4.1- Hardware Platform - Higgs)

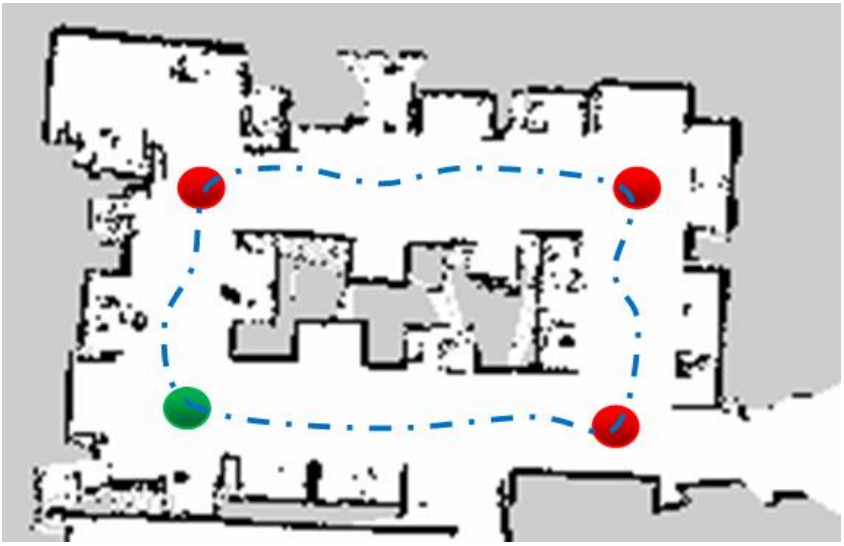


Figure 5.2: Higgs surveillance path in *Sala de Calculo*. The green circle represents Higgs's initial position.

Higgs may use the following hardware systems (mentioned in 4.1 Hardware Platform - Higgs) as sensors, to localize itself and to detect obstacles:

- *Laser (primary sensor for point scan)*
- *Kinect (secondary sensor for point scan)*
- *Arduino (Compass, accelerometers, battery sensors)*
- *Odometry (provided by the base platform Pioneer 2-AT8)*

Higgs will also use the base platform Pioneer 2-AT8 to move and to supply power (through the batteries) to all the onboard systems, except the laptop which use its own battery.

At some point during the mission, there will be a simulated failure of the laser which would make this sensor unavailable. Higgs must be able to recover from this failure, reacting autonomously to it by reconfiguring its control architecture to use the remaining sensors so as to continue with the mission.

Therefore, Higgs's main goals for the mission are:

- Higgs must navigate through the rectangular corridor of *Sala de Calculo* while avoiding obstacles.

- Higgs must localize itself accurately;

- Higgs must recover from a laser failure (that could happen at any time during the mission) with a proper reconfiguration that enables the continuity of the mission.

Some other requirements for the mission:

- Higgs must have some world model elements such as a map of *Sala de Calculo* and the basic waypoints which define the surveillance path (green circles and red circles in Figure 5.2). Higgs must also know its dimension and its initial position in the map must be specified at the start of the mission.

- Higgs must be able to handle some degree of uncertainty in the environment, such as unknown obstacles, sporadic people passing by, etc.

6 Higgs's Control Architecture

In this chapter, the designed Higgs's control architecture for the mission is going to be described. The control architecture is essential for the execution of the mission since it integrates all Higgs's sensor and actuator in an organized schema which enables Higgs to use them adequately. The developed meta-control layer (also part of Higgs's control architecture) is also going to be detailed in this chapter, as for its integration in the architecture. This latter is essential for the fault recovery that must occur when Higgs's laser fails. Without the meta-control layer, when this fails occurs the control system will crash and Higgs will be unable to continue its mission. However, firstly, the requirements for the architecture are going to be mentioned, which served as guidelines for the design of the architecture.

6.1 Requirements for the Architecture

The main requirement for the architecture is that it must cover Higgs's mission described in the previous chapter 5. That said, the system associated with this architecture must be able to control the behavior of Higgs properly in order to make him execute successfully the mission.

It must be structured in modular components, each of which with different responsibilities. This way, the inclusion or substitution of components is made more easily. The meta-control components, part of the meta-control layer, have to be separated from the other base components. This makes it easier to change the meta-control layer of the architecture. This requirement is essential if we wish to implement meta-control into another system, applying the same methodology and using the same meta-control architecture.

Regarding the base control architecture, it is important that it respects the available hardware systems to avoid the presence of components in the architecture which cannot be implemented in the system afterwards. Despite this, the architecture must be sufficiently generic so as to be easily reused in other mobile robots, with a slight adaptation of the new available hardware components.

The architecture must specify a system with self-awareness and self-reconfigurable capabilities, which can monitor and manage itself. It must be easily extendable to handle other failures or events, and must be made compatible with other systems.

6.2 Overview of the Developed Architecture

Having in mind the requirements mentioned in the previous section, a control architecture for Higgs has been designed. It is composed by functional components which have specific tasks and that communicate with each other. Their functions, interactions and organization enables Higgs to perform its surveillance mission. This

control architecture integrates Higgs's sensor and actuators. It also contains representations of the world model and Higgs's model. Generally, Higgs's full control architecture developed for this work can be decomposed in two parts (or layers):

- **The base control architecture** (also referred as: control architecture): this constitutes the control architecture of Higgs responsible for the surveillance mission without the necessity of handling any fails or reconfigurations. It is somehow rigid, in the sense that it is not able to reconfigure itself, nor change its internal parameters. The overview of this architecture is going to be described in the section 6.3. Its functional components will also be detailed in that section.

- **The meta-control architecture:** this specifies an additional meta-control layer that is responsible for managing the base control system. It can reconfigure the base control system (specified by the base control architecture) and change its parameters when special events happen, such as hardware/software failures. Without this layer, Higgs would be unable to recover from the failure that will occur during the mission. This meta-control will be detailed in 6.4.

Higgs's control architecture can be categorized as a hybrid architecture, since it has reactive functional components (such as the navigator which has to avoid some dynamic obstacles that may appear), as well as deliberative elements (such as the long-term planning of the surveillance path that has to be executed). However, with the meta-control layer, this control architecture can also be considered as cognitive, since some aspects of introspection and self-awareness appear to exist when the architecture specifies components that can analyze and reconfigure the system itself. It is also important to mention that the meta-control layer adds some fault-tolerance capabilities to the system, since it will be especially useful (in the specific context of the implementation of this work) for a laser error recovery.

6.3 Higgs's Base Control Architecture

The diagram for the designed Higgs's control architecture can be observed in Figure 6.1. The blue components represent functional components. The red components represent hardware systems (already mentioned in 4.1 Hardware Platform - Higgs). Finally, the green modules represent data that store information necessary by the functional components.

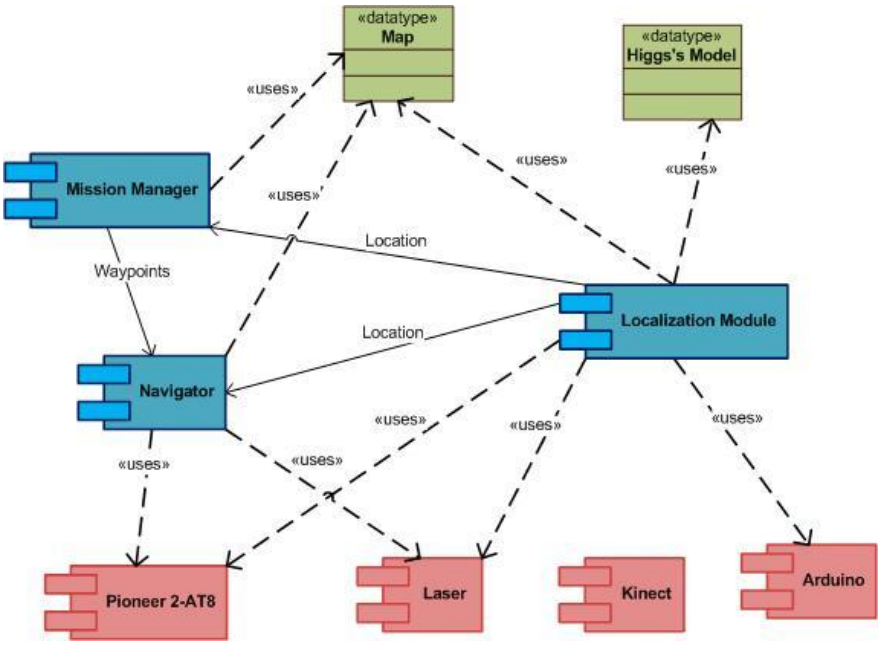


Figure 6.1: Diagram of Higgs's base control architecture

Higgs's Model contains all the relevant information (regarding the robot) needed by the functional components to execute the mission. For the particular mission of Higgs, some of this important information can be Higgs's dimensions, the position of the sensor with respect to the mass center of Higgs, and some aspects of Higgs's cinematic model. The data type *Map* can also be interpreted as the *World Model*, in the way that it constitutes the knowledge of the robot's environment as perceived by the robot. For this particular mission, this *World Model* includes only the map of the environment (*Sala de Calculo*), which must be previously obtained offline.

As for functional components:

- **Mission Manager:** this component is responsible for giving the waypoints to the Navigator module so as to make the robot move in the desired way, i.e., making laps through the rectangular corridor of *Sala de Calculo*. This component must know the location of the robot (given by the Localization Module), the map in which the robot navigates and the basic waypoints which define the desired motion path. When it sends a goal position to the Navigator component, it monitors the location of the robot waiting for the arrival to the goal position. When the robot arrives, the Mission Manager sends the next waypoint to the Navigator as a new goal position. Making this continuously and iteratively makes the robot move in a closed loop through the corridor of *Sala de Calculo*.

- **Navigator:** this component is responsible for generating proper navigation instructions that will be sent to Higgs's base platform (Pioneer 2-AT8). It needs the location of the robot (given by the Location Module) and the map to generate the needed instructions of a proper navigation to the goal position previously set by the Mission Manager. The Navigator component is also responsible for obstacle avoidance, thus it needs the point scan (that can be provided by the Laser or by the Kinect) for obstacle detection and Higgs's

dimensions and kinematic model (part of Higgs's model) for a proper obstacle avoidance and navigation. The Navigator is composed of a global (high level planning) and a local planner (low-level navigation and obstacle avoidance), as referred in section 2.4.2 - Robot's Motion Planning and Navigation.

- **Localization Module**: this component is responsible for computing Higgs's position in the map. It is an essential component in Higgs's control architecture because without knowing its location, Higgs is unable to navigate. As primary sensor data it uses a point scan (that can be provided by the Laser or by the Kinect) which it maps with the landmarks in the map in which the robot navigates. It also uses the odometry (provided by Higgs's base platform) to estimate the robot's pose, which can be improved by integrating it with the compass data (provided by the Arduino board) with an EKF filer. This module also needs some information regarding the location of the Laser and the Kinect device with respect to the mass center of the robot, in order to reference the point scan to the robot's coordinate frame. This latter information is retrieved by the Localization Model from the *Higgs's Model*.

It is important to mention that the hardware systems (represented by the red modules in Figure 6.1) not only correspond to the respective hardware devices but also include their driver components, which enable the access to these hardware devices by the other software components.

This control architecture shows itself as relatively simple and apparently standard. However, it respects all the requirements mentioned in the section 6.1 regarding the base control architecture. It is modular - constituted by several components, each with different responsibilities and tasks exchanging data/message with each other through interfaces. It respects all the available hardware systems because each of the red components (Pioneer 2-AT8, Laser, Kinect and Arduino) are associated to an available hardware device (see 4.1 Hardware Platform - Higgs). It is also sufficiently generic to be applicable to another mobile robot, if this robot has the essential hardware devices needed for autonomous localization and navigation. In that case, to adapt the architecture, it is only needed to insert the proper hardware components. However, this architecture is not sufficient to specify a control system that enables Higgs fulfill its surveillance mission. This is due to the fact that it is not capable of recovering from the laser error, which constitutes one of the most crucial aspects of Higgs's full mission.

6.4 Meta-control Architecture

As you can observe, in the developed base control architecture, described in the previous section, the hardware system *Kinect* is not used by any functional component (see Figure 6.1). This is due to the fact that the *Kinect* hardware system outputs the same data type of the *Laser*: a point scan[7]. Therefore, this makes *Kinect* an apparently redundant hardware system, since it does not add useful information to the robot's

[7] A point scan consists in a n radial readings that represent points in the world with their correspondent distances to the robot

control system because it can be used transparently as a laser. However, having in mind that, during the mission, a laser error will occur, making this device unavailable (as described in chapter in 5 – Higgs's Mission), the *Kinect* may be more useful than it may appear. In fact, after the laser fail, it will become the solely responsible of providing point scans to the system. However, the base control architecture's components are not able to reconfigure the whole system to use the *Kinect* sensor when the laser fails. Therefore, a meta-control layer is crucial for the error recovery and necessary for the Higgs's Mission. This layer will manage the base control system, monitoring it and actuating over it whenever necessary. In this section, the developed meta-control architecture is going to be described and the full architecture is going to be compared with some of the other architectures already mentioned in this document.

The meta-control architecture, developed for this work, is constituted by three main elements, which can be observed in Figure 6.2:

- **A meta-monitor**: this element monitors the control system in search of possible messages, errors or other events that can represent special events for the meta-controller to handle. When detected any of this events, it will alert the meta-actuator (described below), which handles this event, reconfiguring or readjusting the control system if necessary.

- **A meta-actuator**: this element actuates over the control system when a special event is detected by the meta-monitor. It has total authority to manage the control system as it wishes, in the sense that meta-actuator can send/receive messages to/from any component and hardware system, it can start/stop the execution of functional components and can even interrupt or shut down the entire system.

- **Meta-model**, which encompasses**:**
    - **System's configuration**: this stores all the information relevant for the meta-controller. It is updated by the meta-monitor and used by the meta-actuator. It stores the control system's state –current goal– which is used by the meta-actuator when, after handling a critical reconfiguration, it loads the previous system's configuration.
    - **Events:** These consist of meta-events that are associated with system's specific situations in which the meta-control layer has to reconfigure it. They are detected by the meta-monitor (which will trigger them) and by the meta-actuator (which will handle them).

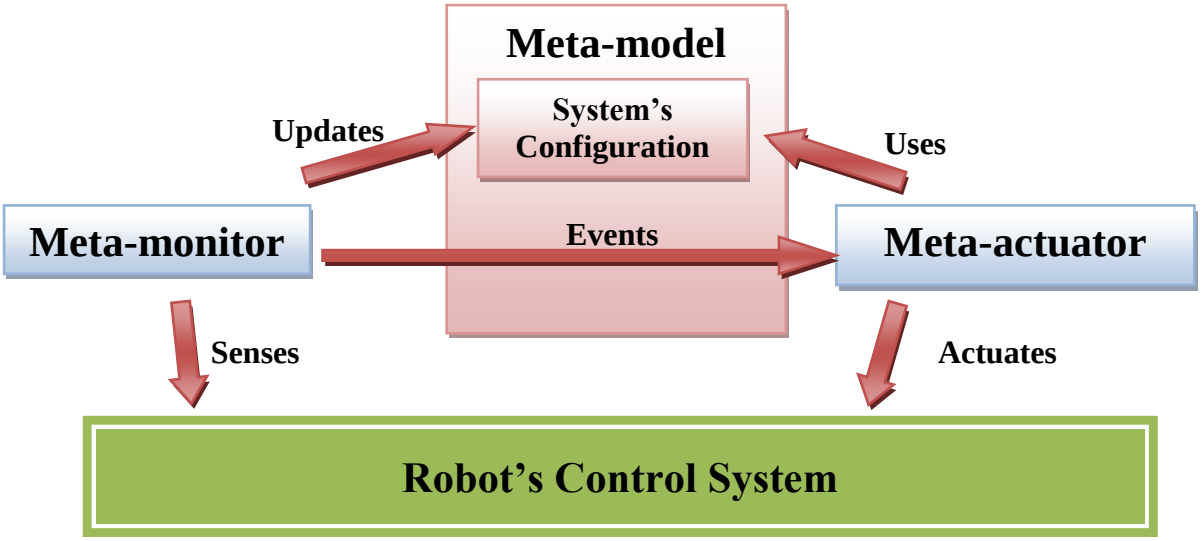


Figure 6.2: Meta-control architecture

The designed meta-control architecture is similar to fault-tolerant control architecture reported in (Blanke, et al. 2006), already summarized in 2.3 - Fault-tolerance in Autonomous Systems and presented in Figure 2.10. The meta-monitor corresponds to the diagnosis block in the architecture presented in (Blanke, et al. 2006), while the meta-actuator corresponds the the re-design block which uses the fault information and adjusts the controller to the faulty situation.

As you can observe in Figure 6.3, the meta-control architecture, along with the base control architecture, follows closely the general architecture *The Operative Mind*, defined in (Hernandez, Lopez and Sanz 2009) which was already described in section 3.3 - ASys Cognitive Patterns. In the *The Operative Mind* there are also *meta-nodes* that monitor and control the operation of nodes. They are patterned after the epistemic control loop 2 (see Figure 3.4), which has many resemblances with our meta-controller architecture. Figure 6.3 shows the mapping of this epistemic control loop with the architecture of the meta-control layer used for this work. It shows that the meta-monitor is associated with the functional tasks of *Observe* and *Perceive*, while the meta-actuator attends the *Redesign* and *Reconfigure* tasks. Finally, there is an element in the epistemic control loop 2 that can be mapped to the Meta-Model, which is represented as the *Fuctional Model*. This element also stores all models' relevant information necessary for the meta-action layer. It can be said that the developed architecture corresponds to an instance of the *Operative Mind* architecture described in (Hernandez, Lopez and Sanz 2009), in which the meta-action component (meta-actuator) receives directly messages by the meta-perception component (meta-monitor), and so the Functional Model is implicit.

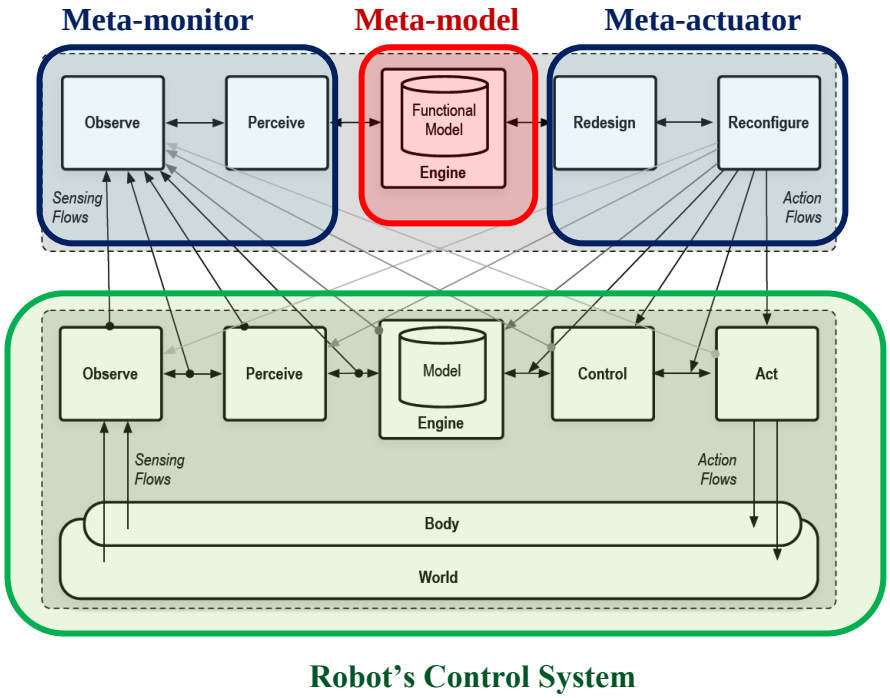


Figure 6.3: Mapping of the epistemic control loop 2 of The Operative Mind with the developed meta-control's architecture

In Figure 6.4, the activity diagram of the meta-actuator is presented, which shows an overview of the execution stages of the meta-actuator when an event is signaled by the meta-monitor. As you can see, after noticing an event, the meta-actuator immediately interrupts the system. This is important to control error propagation through the system because. If this is not made, when handling an event that implies a system's reconfiguration, the system's instability may affect negatively the execution of the mission or even the system itself. Therefore, one must avoid keeping the control system active while the meta-control is managing the system. In the particular case of Higgs's mission, interrupting the system means halting Higgs's movement right when the laser error is detected. This is important because the time interval between the laser error and the full integration of *Kinect* in the system may be sufficient for the Localization Module to fail so that no position estimation will be available in the system anymore. After finishing the handle of the event, the meta-actuator may re-activate the system, now that it assumes that the system is stable. The execution stages of this meta-actuator resemble the four-phases of fault-tolerant artificial system as defined by (Jalote 1994), which were mentioned in 2.3 - Fault-tolerance in Autonomous Systems. The *Error Detection* phase corresponds to the event detection; the *Damage confinement and assessment* phase corresponds to the System Interruption stage; the *Error Recovery* phase corresponds to the meta-actuator's Handle Event stage; finally, the *Fault Treatment and continued service* phase corresponds to the System Reactivation stage. In

fact, the meta-actuator execution stages were inspired by this Jalote's phases for fault-tolerant artificial system, which gave an important emphasis on the error propagation avoidance (that in our whole system is handled by a system temporary halt). Nevertheless, as already mentioned in 3.4 - Beyond Current State of the Art, adding the meta-control layer does not makes the system exclusively fault-tolerant because it is not only *fault-reactive*. The events can correspond to special non-faulty occasions in which modification in the control system can upgrade the efficiency of its mission/task execution. Hence, the specified control architecture (with its meta-control layers) can be considered more as a cognitive architecture, because it presents (though primitive) self-awareness capabilities.

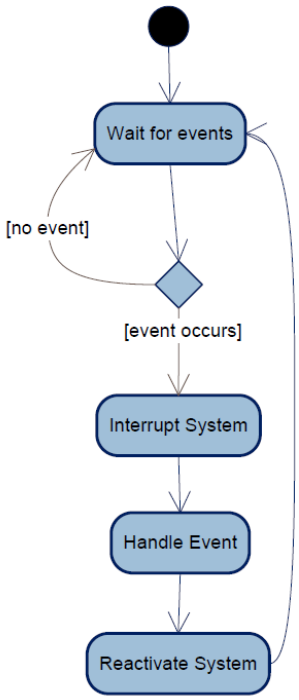


Figure 6.4: Activity diagram of the Meta-actuator

6.5 Meta-control Architecture Integration in the Base Control Architecture

In Figure 6.5, a diagram of Higgs's full architecture for the mission is presented. As you can see, and compared to the base architecture presented in Figure 6.1, the meta-control layer is added, with its respective components (Meta-monitor, Meta-model and Meta-actuator).

As said before, the meta-control layer's components can monitor and manage the control system, which adds introspection and self-reconfiguration capabilities to the overall system. These capabilities are crucial for the success of Higgs's mission because the system will need to be reconfigured automatically when the laser fails, engaging the *Kinect* hardware system to be used by the system's components.

One key aspect of the meta-control layer is that it is attached to the base architecture without the necessity of a redesign of the latter. Thus, the base architecture defines a system that is operational most of the time except when the events that are associated with the meta-control are triggered (that is, the laser error in Higgs's mission). In this way, the general methodology used for this specific architecture could be reused for an integration of a meta-control layer to a working and fully developed control system in which we intend to add introspection and self-reconfiguration capabilities or even make the system fault-tolerant, as long as the system is designed in modules that provide information on their state and an interface for their managing. Summarily, the meta-control layer could be added to "upgrade" the system's robustness without the necessity of a system's redesign. This establishes one of the main advantages of the approach for meta-control integration that was used for this work.

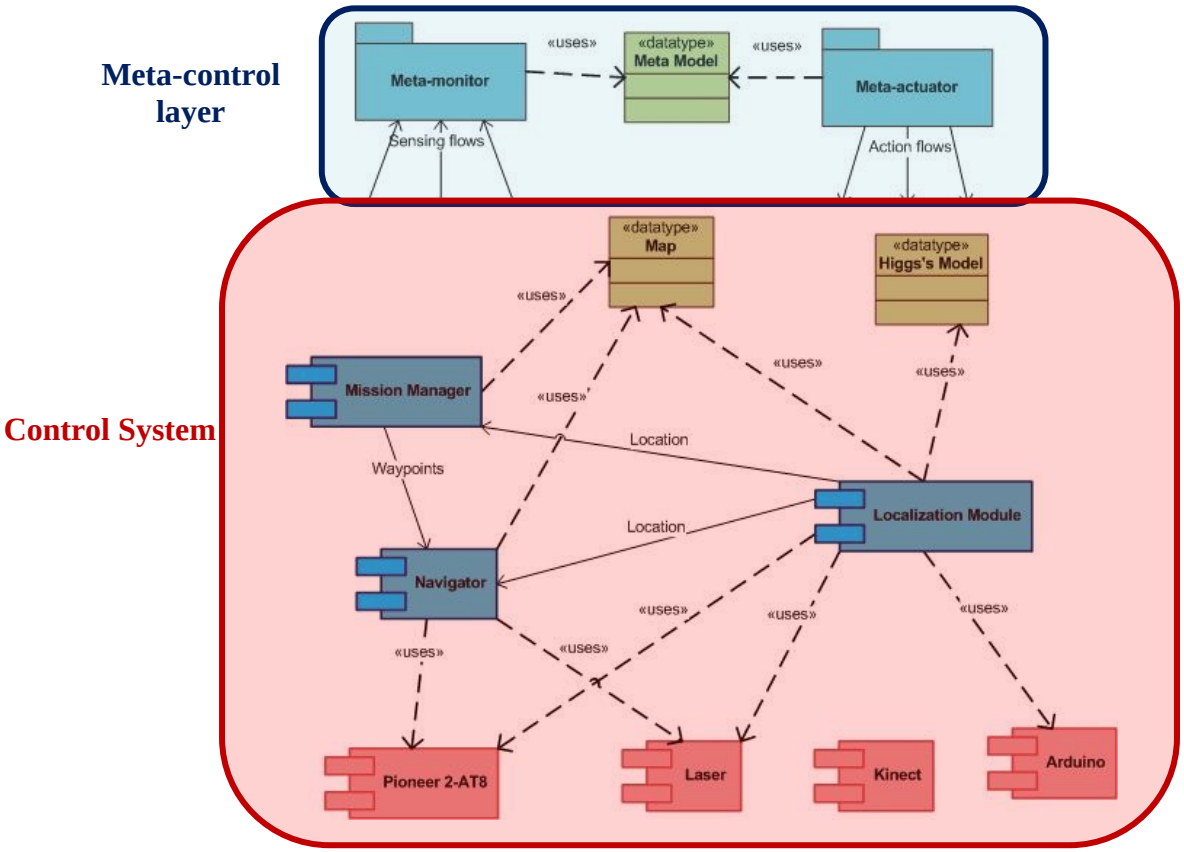


Figure 6.5: Diagram of Higgs's full architecture control system, with the meta-control layer

7 Implementation

In this chapter, software implementation of the Higgs's control system is going to be described, which follows the designed architecture presented in the previous chapter 6.

The implementation of Higgs's control system was made over the robotic middleware ROS (described in the section 4.2). ROS was chosen over other robotic software platforms because it brings many advantages that are suitable for the intended control system:

- It presents a modular structure with executable nodes, which can be mapped with the modular components defined in our developed architecture.

- Each node provides several interfaces (topics, services, parameters), that can accessed remotely.

- It provides many algorithms and drivers for autonomous mobile robots that can be reused.

- It provides many useful utilities to monitor the nodes and the control system.

In the first section, the ROS nodes associated with the base control system are going to be described, with a global overview of their functionality and integration. In the second section, the meta-control layer's ROS nodes are going to be explained and their integration in the base control system is described in the third section.

7.1 Base Control System

In this section, the implementation associated with the developed Base Control System for Higgs's is going to be described, as for the ROS nodes used and reused.

7.1.1 Drivers and Low-Level Nodes

In this section, the implementation details of the nodes that were used as drivers for robot's devices are going to be described, as well as low-level nodes which integrate sensor's data.

7.1.1.1 Base Platform - Pioneer

As a driver for the Base Platform (Pioneer 2-AT8), the ROS package *pioneer_aria* was implemented, based on the ROSARIA package developed by Srećko Jurić-Kavelj that is available in the ROS repositories. This package contains the implementation of a ROS node (called *pioneer*) that consists in a ROS wrapper for the MobileRobot's ARIA C++

library. It enables to set velocities and read odometry data from the Pioneer base platform (see Figure 7.1).

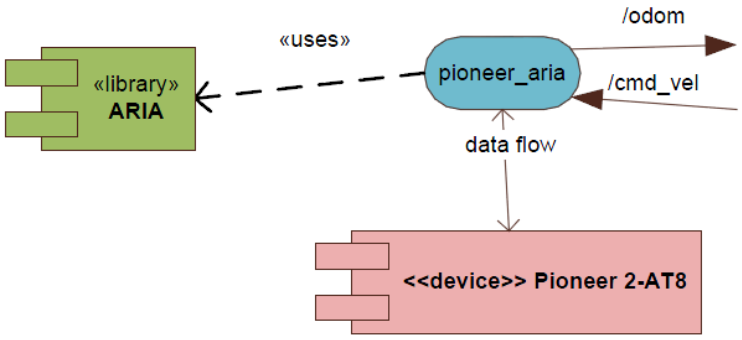


Figure 7.1: The ROS node of the *pioneer_aria* package

The ROS Node *pioneer* was implemented in C++. It publishes to ROS topics the odometry read from the robot platform (ROS topic */odom*) and the sonar data (ROS topic */sonar*). It is also subscribed to the ROS topic */cmd_vel* from which it listens to linear and angular velocity commands.

7.1.1.2 Laser

For the laser driver, the ROS package *sicktoolbox_wrapper* was used. It is available in the ROS repositories and maintained by Jason Derenick of the GRASP Laboratory at the University of Pennsylvania. It consists of a wrapper for the SICK LIDAR Matlab/C++ toolbox library[8]. This latter offers stable and easy-to-use C++ driver for the laser Sick LMS-200, which is the one that Higgs uses as a laser device. The ROS node implemented in this package publishes, to the ROS topic */scan*, the laser point scan as a *LaserScan* message type. This message type is defined in the common package *sensor_msgs*. Therefore, any other ROS node interested in getting the laser scan, only have to subscribe to the mentioned ROS topic.

7.1.1.3 Kinect

As a driver for the Microsoft's Kinect sensor, the ROS package *openni_camera* was used. This package is available in the ROS repositories and was made by Suat Gedikli, Patrick Mihelich and Radu Bogdan Rusu. This driver consists of a ROS node that publishes raw depth, RGB and IR image streams.

To use the Kinect as a simulated laser, we must only consider the depth streams as relevant data. This corresponds to a 3D Point Cloud that is of ROS message type *PointCloud2*, defined in the common package *sensor_msgs*. However, for a client node to use transparently and equally data from the laser and from the Kinect, the point scan must have the same message type. Therefore, a few ROS nodelets were added to the system as wrappers to convert the *PointCloud2*, generated by the Kinect, to a

[8] Available in http://sicktoolbox.sourceforge.net/

LaserScan. This way, both devices' drivers (of Kinect and Laser) outputs the same data type to the system and the replacement of one by the other is made more easily.

To convert a 3D Point Cloud of message type *PointCloud2* (generated by the Kinect's driver) to a laser scan of message type *LaserScan*, we used the ROS package *pointcloud_to_laserscan* of the *turtlebot* stack, available in the ROS repositories and developed by Tully Foote. This package was specially developed for making devices like the Kinect appear like a laser scanner for 2D-based algorithms (e.g. laser-based SLAM). This package contains the implementation of two ROS nodelet[9] (*CloudThrottle* and *CloudToScan*). These nodelets are linked to the data published by the *openni_camera* node via the nodelet *openni_manager*, implemented in the *openni_kinect* ROS stack.

The conversion from a *PointCloud2* to a *LaserScan* is made in two steps, each of which is associated with the two Nodelets implemented in the *pointcloud_to_laserscan* package:

- Transformation of the 3D point cloud to a throttled point cloud: this is done by the ROS nodelet *CloudThrottle* of the *pointcloud_to_laserscan* package, which compresses the *PointCloud2* into a smaller structure, without any data loss. This ROS nodelet receives *PointCloud2* data via the *openni_manager* nodelet, and then returns the throttled cloud to the *openni_manager* which publishes the throttled point cloud to the topic */cloud_throttled*.
- Transformation of the throttled cloud to a laser scan: this is done by the ROS nodelet *CloudToScan* of the *pointcloud_to_laserscan* package. This nodelet basically receives a throttled cloud from the *openni_manager*, transforms it to *LaserScan* message data, and returns it to the *openni_manager*, which publishes it to a topic. We can set the maximum and minimum height of the points it the throttled point cloud to be considered for the laser scan. This way, we can manage the vertical angular range of the Kinect as we wish. The computational cost of this nodelet is far inferior to the one of *CloudThrottle* nodelet.

The advantage of separating these processes into a "*PointCloud2* → throttled cloud" and "throttled cloud → *LaserScan*" is that we can have in our system a single *CloudThrottle* nodelet that is used by two or more *CloudToScan* nodelets. This is useful when we want to publish two or more laser scan derived from a single point cloud published by the Kinect, each with different angular ranges, without adding a considerable computational cost to the system. In our system, we use two *CloudToScan* nodelet: one that provides a wider scan used for obstacle avoidance, received by the navigation node, to detect tall and short obstacles; the other that provides a narrower scan, received by the localization node, which is better to simulate the laser scan which is made in a single horizontal plane. These both *CloudToScan* nodelet use the same *CloudThrottle* nodelet.

An overview of the Kinect driver can be seen in Figure 7.2. The red component represents the Kinect device. Blue processes are ROS nodes and the green processes are ROS nodelets.

[9] For in information on ROS nodelet, see section 4.2.3 - ROS Client libraries and Nodelets

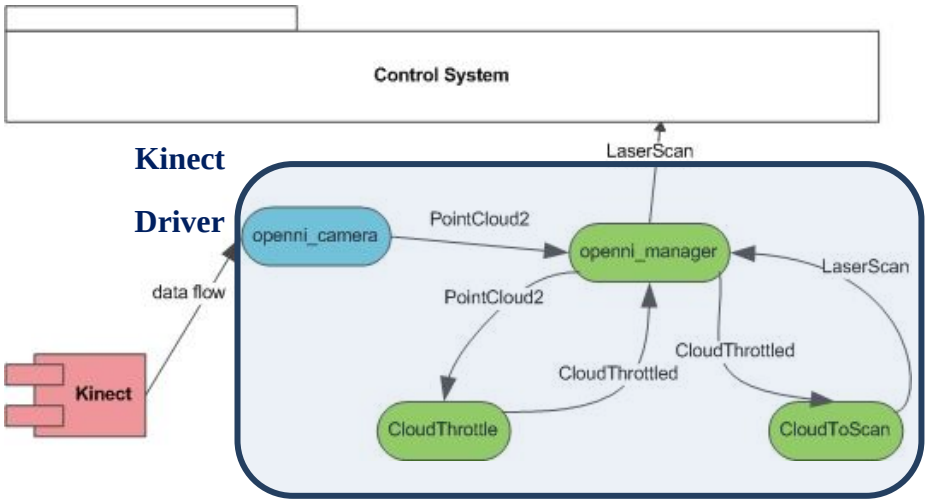


Figure 7.2: Elements of the Kinect driver in the system.

7.1.1.4 Arduino

For the Higgs's Arduino board, we reused the driver implemented by Marcos Garcia (a former member of the ASLab group), which is described in his report (Garcia 2009). This driver was implemented as a CORBA[10] servant, using the CORBA implementation TAO[11].

Due to the stability of this driver, instead of implementing an Arduino driver from scratch, we implemented a "CORBA to ROS" wrapper that reuses the mentioned CORBA driver. As a result, the ROS package *higgs_arduino_corba_ros* was developed. This package contains a C++ implementation of a ROS node with the same name, that is also a CORBA client. It communicates with the CORBA servant using the CORBA communication protocol and retrieves data obtained by the Arduino board's sensor. For the implemented control system, the only Arduino's meaningful data is the compass readings. This compass reading is obtained by the ROS node and is published as an *IMU* (Inertial Measurement Unit) message type to the ROS topic */output_Imu*. The *IMU* message type is defined in the ROS package *sensor_msgs* and holds data regarding the orientation of an object, as well as its angular and linear velocity, with the respective covariance matrices. This message type is used in ROS to encapsulate data read from digital compasses, accelerometers and gyroscopes.

An overview of the Arduino's driver elements can be appreciated in Figure 7.3.

---

[10] CORBA (Common Object Request Broker Architecture) is a standard, which specifies a system that enables transparent interoperability between systems that run on heterogeneous and distributed environments. Its design is based in the object model of OMG (Object Management Group), which defines the external characteristics of objects that can operate on different implementation forms. For more information on CORBA

[11] For more information on TAO, see http://www.cs.wustl.edu/~schmidt/TAO.html

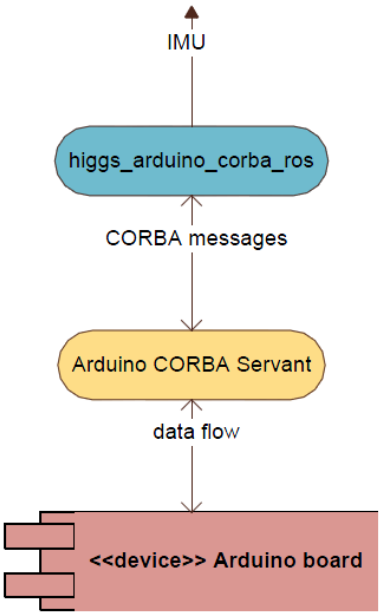


Figure 7.3: Arduino's full driver elements

7.1.1.5 EKF for Pose Estimation

An EKF (Extended Kalman Filter) was used to integrate the compass reading (retrieved by the Arduino board) and the odometry (retrieved by the base platform Pioneer 2-AT8) into a combined odometry. The integration of the compass reading with the odometry using an EKF improves considerably the accuracy of the predication of the robot pose. This is because the odometry error rises significantly as the robot's angular speed increases. Using the compass reading can lower or even nullify the robot's orientation error and helps the localization node to make better predictions of the robot pose.

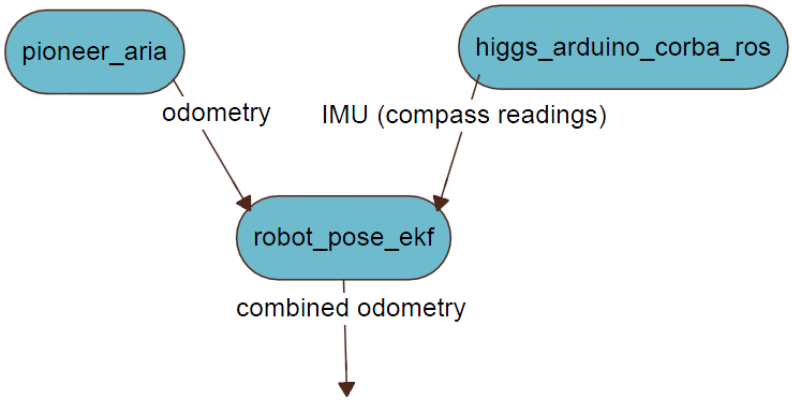


Figure 7.4: Integration of odometry and compass reading through an EKF

For the EKF, we used the ROS package *robot_pose_ekf*, of the *navigation* stack, available in the ROS repositories. It was developed by Wim Meeussen and it is

generally used to estimate the 3D pose of a robot, based on (partial) pose measurements coming from different sources. It uses an extended Kalman filter with a 6D model (3D position and 3D orientation) to combine measurements from wheel odometry, IMU sensor and visual odometry. In our system, the *robot_pose_ekf* node will subscribe to topics where the odometry and the *IMU* are published (see Figure 7.4). It then publishes, to a ROS topic, the combined odometry (after passing through the EKF filter) that can be used by the navigation and the localization node as a better estimation of the robot's pose than simply using the odometry data.

7.1.2 World and Robot Model

In this section, the implementation details of the nodes that represent the world and the robot's model are going to be presented.

7.1.2.1 Transformation Broadcaster

A transformation broadcaster is used to broadcast transforms that indicate the geometrical relationship of relevant 3D coordinate frames in our system. The transforms are organized in a tree structure (see Figure 7.5) that is buffered in time, and lets other ROS nodes translate points, vectors, etc. between any two coordinate frames at any desired point in time. They are composed by a 3D translation vector and a rotation quaternion. Transforms are extremely useful in ROS because a robot usually has many 3D coordinate frames that change over time, such as a world frame, base frame, gripper frame, head frame, etc. The transformation broadcaster keeps track of all these frames over time and enables other nodes to get the coordinates of points, vectors and other geometric entities with respect to a specific coordinate frame (e.g., the current pose of the base frame in the map frame).

In our system we implemented a transformation broadcaster that broadcasts static transforms that represent the robot's model, in the sense that they relate the point representation of the robot (that we named *base_link*) with the 3D position and rotation of the available sensor devices. This way, we can reference the sensed data (that usually is natively referenced to the sensor's frame) to the *base_link* frame of the robot. This is needed in our system, for example, to avoid obstacles or for localization, since we must reference the point scan gotten from the laser or the Kinect to the robot's coordinate frame. These transforms (that relate the robot's frame to the device's frame) are usually static, i.e. they never change over time, unless the devices move with respect to the robot's base, which is not Higgs's case. These static transforms represent Higgs's (static) model in our system.

A ROS package named *higgs_setup_tf* was implemented. This package contains the implementation of a ROS node that broadcasts the transforms of *base_link* (the robot's reference frame) to the laser and Kinect's frames. To calculate these static transforms, we previously measured the position and orientation of the laser and the Kinect with respect to the considered robot's frame.

There are other ROS nodes that publish dynamic transformation to the system, such as the transforms that represents the combined odometry (broadcasted by the already mentioned *robot_pose_ekf* node) and the location of the robot (broadcasted by the localization node that will be detailed in section 7.1.3). Thus, in ROS, the robot's location at a given point in time can be specified by the transforms from the map's frame to the robot's frame.

In Figure 7.5, you can observe the transformation tree of our system, which shows the references frames (the nodes in the figure) and the transforms published in our system (the edges in the figure).

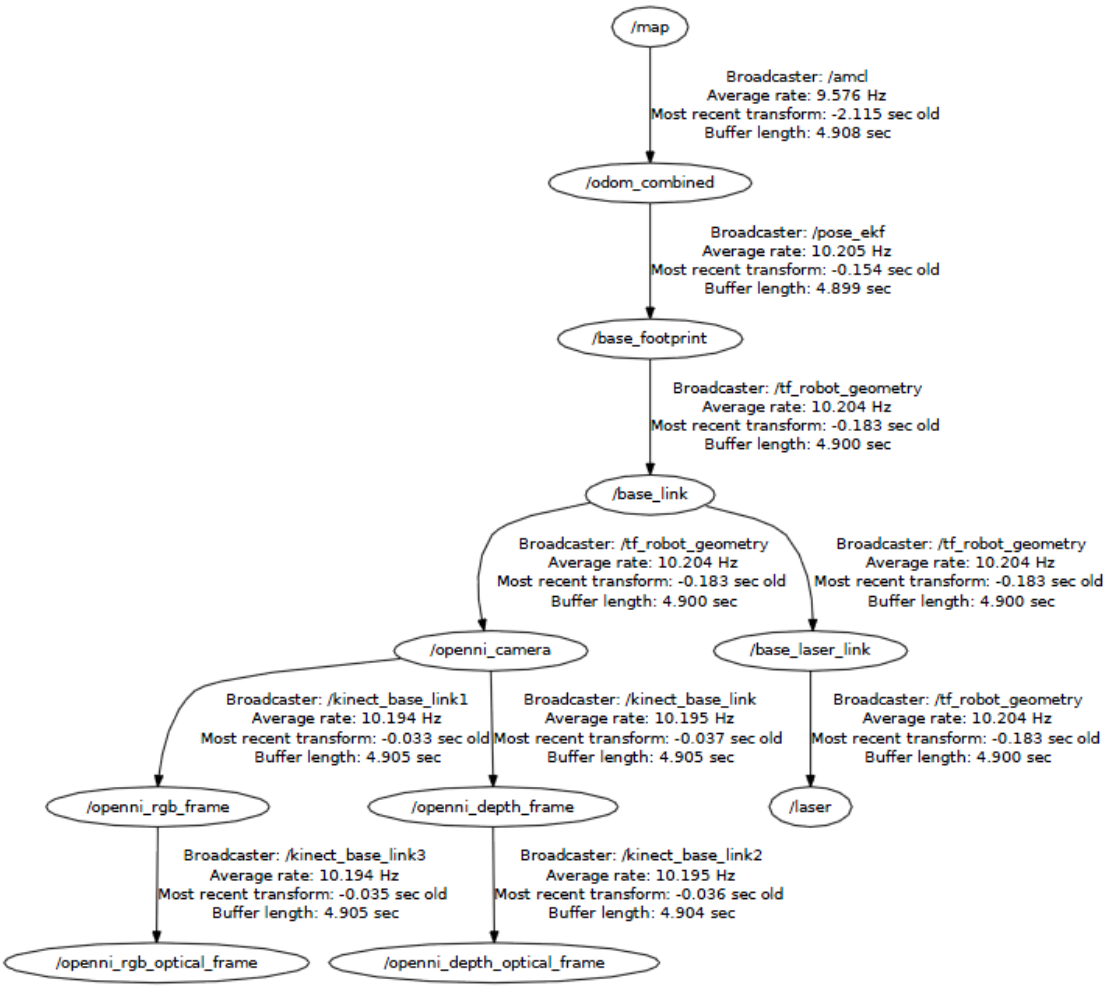


Figure 7.5: Transformation tree of the developed system

7.1.2.2 Map Server

For the implemented system and for Higgs's mission, the World Model corresponds to a 2D map of the environment in which Higgs will navigate, that is, the map of the *Sala de Calculo* of the DISAM (already mentioned in 5 - Higgs's Mission). That said, the map will be available by including a ROS node into the system – *map_server*. This node's implementation is located in the ROS package of the same name – *map_server* – developed by Brian Gerkey and Tony Pratkanis, available in the ROS repositories. The *map_server* loads the map data from pair of files located in the filesystem and publishes, into a pre-determined topic, the occupancy grid that corresponds to the map. The files, that define a map and that are loaded by the *map_server*, are:

- An YAML file that describes the map's meta-data (such as the resolution, 2D origin of the map), and names the image file.

- An image file that describes the occupancy state of each cell of the world in the color of the corresponding pixel. Whiter pixels are free, blacker pixels are occupied, and pixels in between are unknown. In Figure 7.6 you can see an example of an image file that represents an occupancy-grid map.

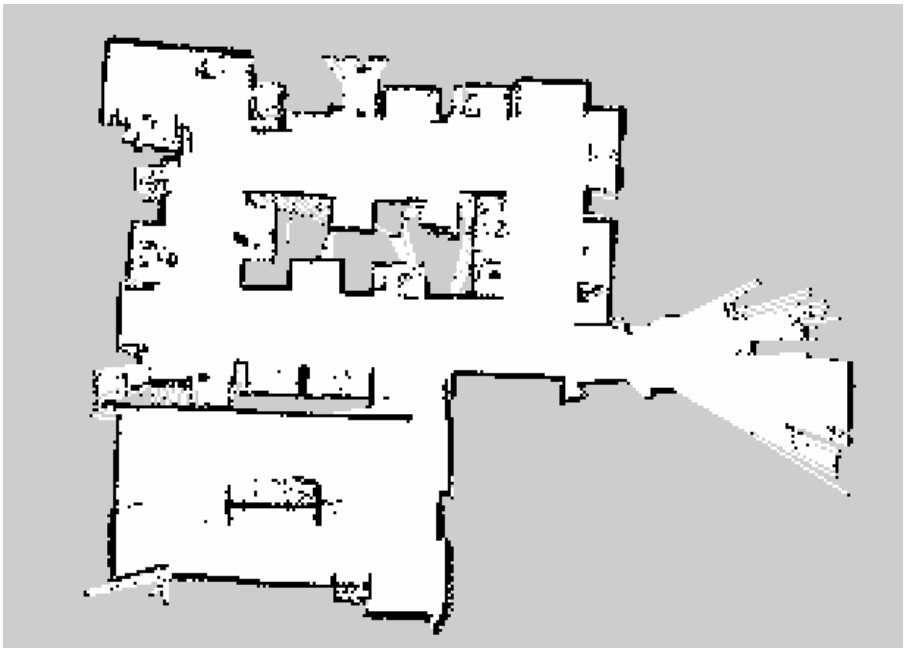


Figure 7.6: An occupancy-grid-based map published by the map_server

This package also provides a command-line utility (*map_saver*) that allows dynamically generated maps to be saved to the respective files. This utility was used for the system's setup, as will be explained later in this document. The package was chosen, mainly, because it is compatible with the localization and the navigation nodes that will be described in subsections 7.1.3 and 7.1.4, respectively. The map is essential for the

operation of the localization and the navigation, as already shown by the system's architecture (see Figure 6.5).

7.1.3 Localization Node

For the system's localization component we reused an available ROS package, named *amcl*, located in the *navigation* stack in ROS repositories. This package implements the adaptive Monte Carlo localization approach (or KLD-sampling) as described by Dieter Fox in (D. Fox 2003). This approach uses a particle filter (already mentioned in the section 2.4.1 - SLAM) and introduces a statistical method that increases the efficiency of the algorithm. This method, named KLD-sampling, consists in adapting the size of sample sets during the estimation process which bound the approximation error introduced by the sample-based representation of the particle filter. For more information on the algorithm, consult the paper (D. Fox 2003) or the book (Thrun, Burgard and Fox 2005).

The package *amcl* was developed by Brian P. Gerkey and was derived from Andrew Howard's Player[12] driver. The *amcl* node is subscribed to a laser scan, to some transforms messages (that relate the laser frame with the robot and the odometry frame) and a laser-based map, and publishes a pose estimation of the robot (see Figure 7.7) with its respective covariance.

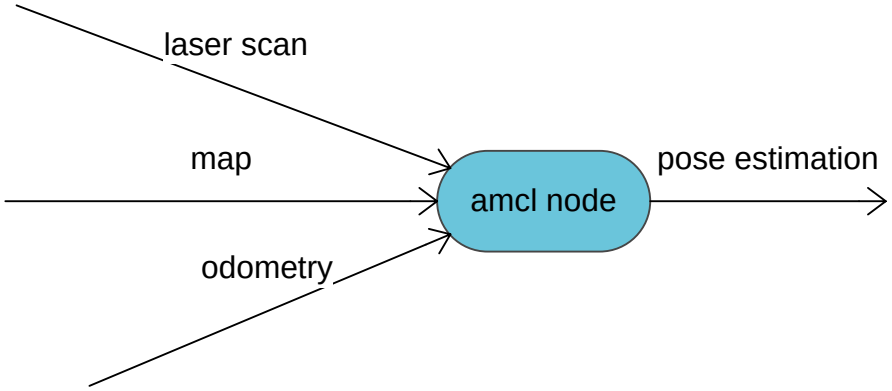


Figure 7.7: *AMCL* node's general inputs and outputs

The *amcl* node is also subscribed to an additional topic used to initialize the position of the robot, i.e., the position of the set of particles. In this topic, the initial pose of the robot in the map is going to be indicated to the system using a graphical interface. This

[12] Player provides free software tools for robotics and sensors. For more information, see: http://playerstage.sourceforge.net/

procedure is going to be explained later in chapter 8 - Experiments and Analysis of the Results.

This *amcl* node has many parameters that need to be adjusted for each robot, environment and sensors. They can be divided in three categories:

- **Overall filter parameters:** These parameters specify the particle filter characteristics (min/max number of particles), maximum error of the distributions, update frequencies, the initial pose and covariance of the particle filter, among others.

- **Laser model parameters:** These parameters define the description of the laser's model, as perceived by the *amcl* node. These include the laser maximum and minimum range, number of beams and measure's covariance.

- **Odometry model parameters:** These parameters are set accordingly to the robot's odometry characteristics. These include the odometry expected noise in the rotation and translation, as well as the frame's names of the odometry, *base_link* and the coordinate frames published by the localization system.

The *amcl* node not only publishes the robot's pose estimation as a position referenced to the global frame (i.e. in the map's coordinate frame), but also as a transformation between the global frame and the odometry frame (see transform from */map* to */odom_combinder* in Figure 7.5). Essentially, this transform corresponds to the drift that occurs using Dead Reckoning. If we add this transform to the one from the odometry frame and the robot's frame (already published by our system's node *robot_pose_ekf*, described in section 7.1.1.5), we have a transform that corresponds to the robot's pose estimation (see Figure 7.8).

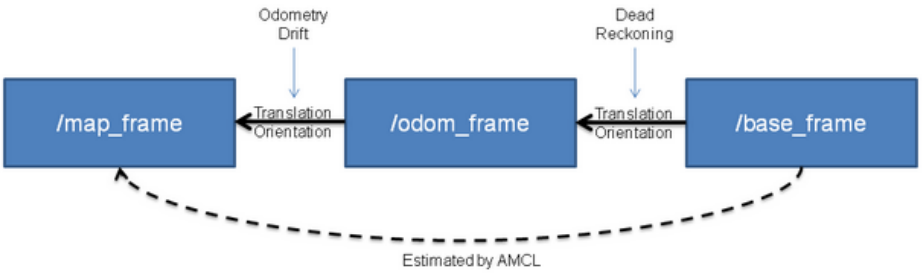


Figure 7.8: Transform for robot's pose estimation, when using the *amcl* node (Osentoski 2011)

In our system, the values of *amcl* node's parameters were set based on the values used by the Willow Garage's robot Turtlebot, which one can find in the *turtlebot* ROS stack available in the ROS repositories. This robot also used the *amcl* node for localization. However, some parameter's values were changed to improve Higgs's pose estimation accuracy. These values were set according to some experiments' results in simple

navigation tasks with Higgs, through a try-and-error method. Some of the parameter's values adjusted were the odometry expected error and the maximum number of particles. These slight value adaptations seemed to improve significantly the localization's accuracy of Higgs using the *amcl* node. Some of the results can be seen in chapter 8 - Experiments and Analysis of the Results.

7.1.4 Navigation Node

In our system, we chose to reuse the *move_base* ROS package, of the *navigation* ROS stack, available in the ROS repositories and developed by Eitan Marder-Eppstein. We found this package suitable for the navigator component of our system because it consists of a well-structured, well-documented and sufficiently generic implementation of a navigation agent in ROS. Furthermore, it is compatible with our localization node – *amcl* - and was already implemented and tested in other systems with successful results. It uses the *actionlib*, a standardized interface that deals with pre-emptible tasks, provided by the *actionlib* ROS package. Therefore, the *move_base* node can be seen as a *actionlib* server, in which, given the goal in the world, its task is to attempt to reach it with a mobile base.

The *move_base* node is constituted by several internal components, such as global and a local planner, and global and local *costmap*, which will be explained below in this section. As inputs, it needs the map in which the robot is navigating, sensor's point scan (such a laser or a point cloud scan), the odometry information, and transforms that relate sensors' and robot's frames. As outputs, it publishes the robot's movement instructions (linear and angular velocity). The *move_base*'s interface and internal components can be appreciated in Figure 7.9.

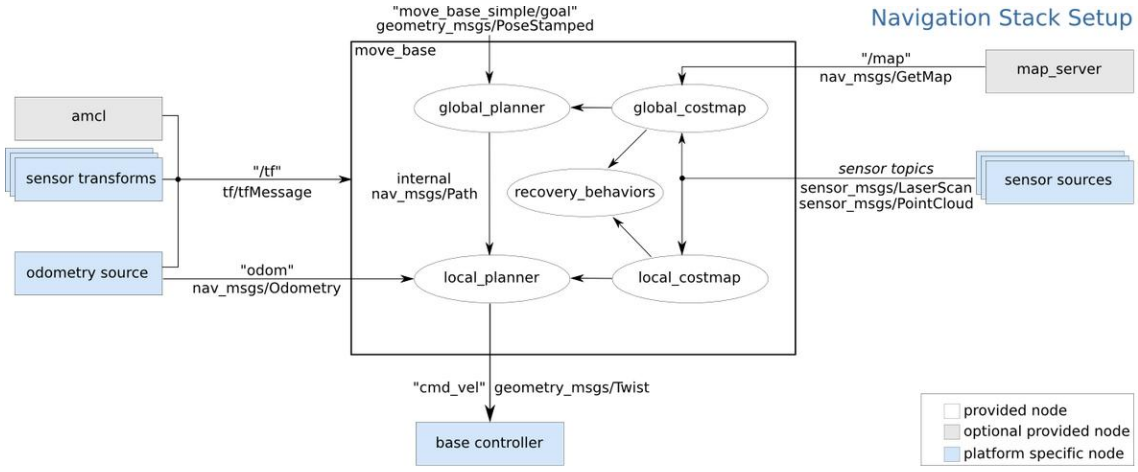


Figure 7.9: The *move_base* node's interface and internal components.[13]

[13] Retrieved from http://www.ros.org/wiki/move_base.

The *move_base* node may perform recovery behaviors, when the robot perceives itself as stuck or lost.

As mentioned above, the *move_base* node is constituted by a global planner and a local planner[14]. These planners can be configured by adjusting their parameters', provided by the *move_base* node. For the local planner, we chose to use a Dynamic Window approach, which is implemented in the *move_base* node together with some other base local approaches. Experimental results showed that this approach provide better results in Higgs's navigation. The Dynamic Window approach was originally specified in (Fox, Burgard and Thrun 1997) and was already mentioned in the section 2.4.2 - Robot's Motion Planning and Navigation. Other parameters of the global and local planner include the max/min accelerations and velocities of the robot, and the goal tolerance.

In the *move_base* node, the global and local *costmap* are used by the global and local planner, respectively. The *costmap* implementation is located in the *costmap_2d* ROS package of the *navigation* ROS stack, also developed by Eitan Marder-Eppstein. The *costmap* takes in sensor data from the world, builds a 2D occupancy grid of the data and inflates costs in a 2D costmap based in on the occupancy grid and a user specified inflation radius (see Figure 7.10). It generally consists in a configurable structure that uses sensor data to store and update information about obstacles and which is the area that the robot must navigate in order to avoid obstacle collisions. In Figure 7.10 we can see the purple cells that represent obstacles, the blue cells that represent obstacles inflated by the inscribed radius of the robot, and the blue polygon that represents the footprint of the robot. To avoid collisions, the footprint of the robot must never intersect the purple cells, and the center point of the robot must never cross the blue cells. The global and local *costmaps* can be properly configured by setting its parameters' values. Some of these parameters define the robot's footprint, the robot's radius and characteristics of the observation sources (point cloud vs. laser scan, obstacle range, min/max obstacles height, etc.).

[14] Consult the section 2.4.2 - Robot's Motion Planning and Navigation, for more information regarding local and global planner

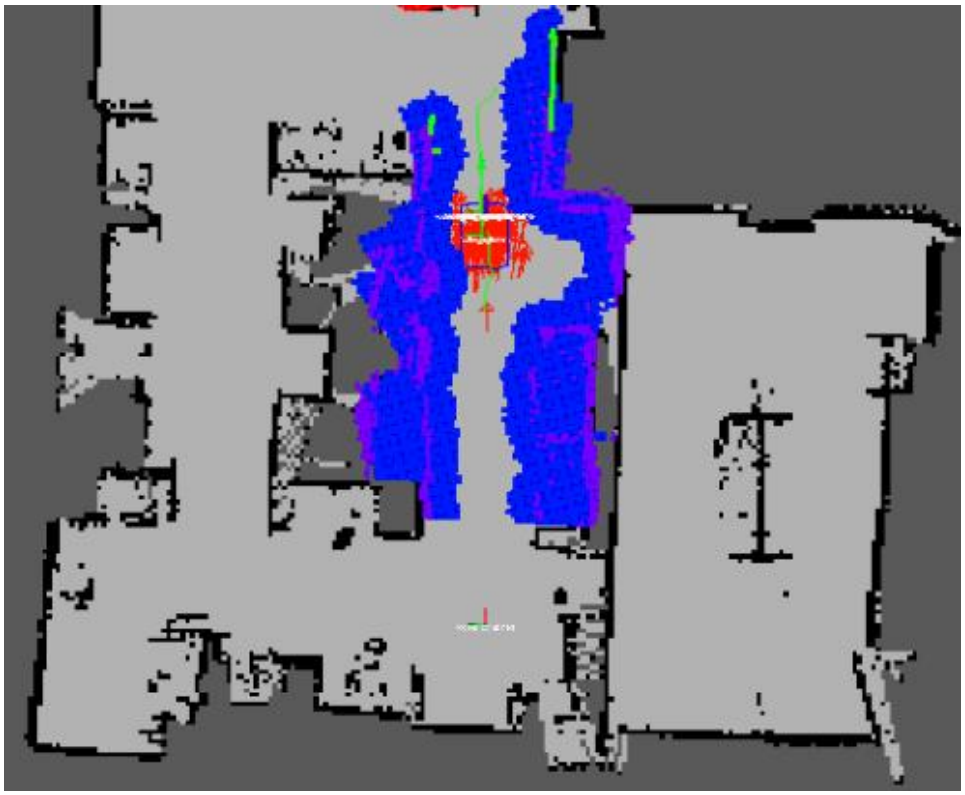


Figure 7.10: The occupancy grid map with superimposed obstacles in purple and inflated obstacles in blue from the *costmap*.

In our system, some values of *move_base's* parameters were set accordingly to Higgs's base platform and devices characteristics. Other were manually set by executing several tests in which we tried to maximize the efficiency of the navigation, analyzing Higgs's performance.

7.1.5 Mission Manager Node

As already mentioned in section 6.3 - Higgs's Base Control Architecture, the mission manager component has the task to set goal positions for the navigator component, in order to make Higgs move through the rectangular corridor of *Sala de Calculo*, in a close loop way. Without this component, the system is not be able to take initiative, and it will wait for external instructions command the movement of the robot.

The mission manager component was implemented as a ROS node. For that, we developed a ROS package – Mission Manager – that contained the implementation of the ROS node *mission_manager*. This node was implemented using *rospy* (python client libraries for ROS), which was already mentioned in section 4.2.3 - ROS Client libraries and Nodelets. Its execution steps can be appreciated in the activity diagram in Figure 7.11. As it can be observed, it first loads the waypoints from a file. This file must be located in the mission manager's package, and must contain a list of waypoints representing the desired route for the robot. Each waypoint represents a position and an orientation in the map's coordinate frame. Therefore, the *mission_manager* node, as we

implemented it, can be reused for other surveillance mission in other environments, as long as we provide the proper waypoints.

The *mission_manager* node, like the *move_base* node[15], uses the *actionlib* standardized ROS interface. However, the *mission_manager* identifies itself as an action simple client that interacts and set tasks to the *move_base* node, which is an action server. Therefore, after loading the waypoints, the *mission_manager* must wait for the *move_base* action server to come up. The *mission_manager* must also wait for the initial localization estimation, set manually in our system through a graphical interface, before starting to set navigation tasks. This is because the localization node is only fully operational after an initial manually set pose estimation. After this, the *mission_manager*, sends goal positions to the *move_base* node (using the *actionlib* interface), starting from the initial waypoint in the list. It waits for the *move_base* to complete its navigation tasks before sending the next waypoint. When the *move_base* informs to the *mission_manager* that the robot has arrived to the last waypoint in the list (completing a full lap), the *mission_manager* repeats the procedure, starting from the initial waypoint in the list. This way, the robot moves in a closed loop way through the route specified by the list of waypoints.

For Higgs's particular mission, since we know our environment before starting the mission (i.e., we have a previously build/obtained map of *Sala de Calculo*), we can also get a set of waypoints that define the route through the rectangular corridor of *Sala de Calculo*. The selected waypoints for the surveillance mission in our system can be seen in Figure 7.12. As it can be seen, they define the route (in red) through the rectangular corridor in *Sala de Calculo*. Before the execution of the mission, the waypoints' coordinates and orientation (in reference to the map's coordinate frame) were calculated and stored the file that the *mission_manager* node loads. It is important to mention that the initial robot's pose, manually set, corresponds to the waypoint 4 in Figure 7.12. Thus, the first navigation task imposed to the *move_base* is to move Higgs's from the waypoint 4 to the waypoint 1.

[15] The *move_base* node corresponds to our navigation component in the implemented system. It was described in section 7.1.4 - Navigation Node.

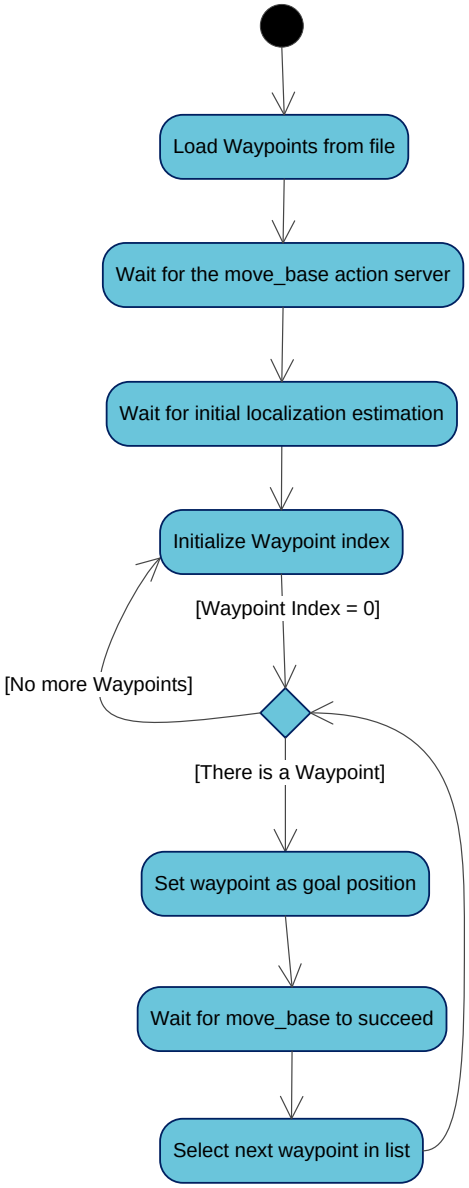


Figure 7.11: Activity diagram of the Mission Manager implemented ROS node

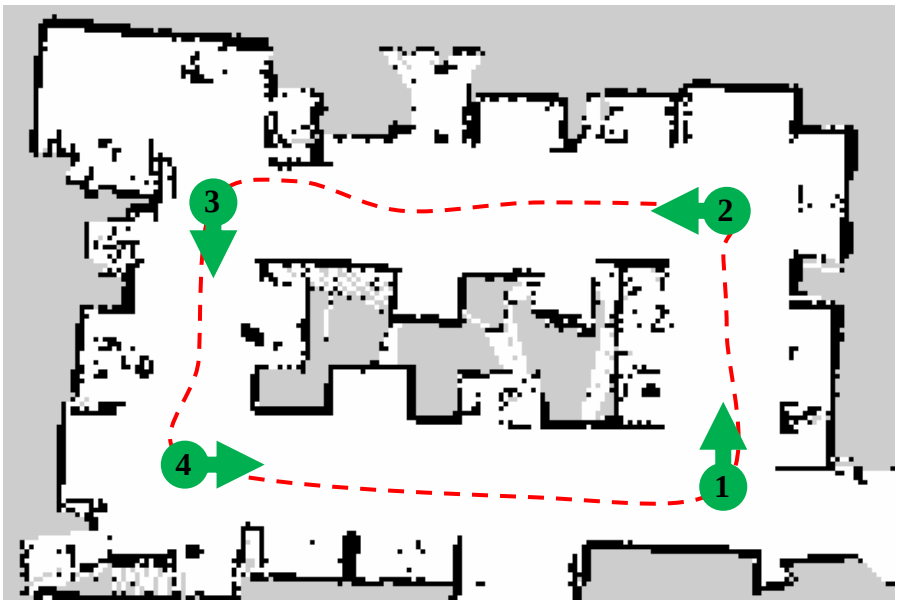


Figure 7.12: Waypoints for Higgs's surveillance mission in *Sala de Calculo*

7.1.6 Base Control System Overview

In Figure 7.13, you can observe the implemented base control system for Higgs's surveillance mission. This graph was obtained using the ROS's *rxgraph* command-line utility, which is used for visualizing the ROS computation graph. The graph corresponds to the implemented base control system in Higgs which is capable of autonomously navigating Higgs's through the rectangular corridor of the *Sala de Calculo*, but it is unable to handle any fail that can occur throughout the execution of the mission (such as a laser fail). For that, and for the self-reconfiguration capabilities, we need to add to the system the meta-control layer, whose implementation will be described in the section 7.2 - Meta-control Layer Implementation

In the graph in the Figure 7.13, the nodes correspond to ROS nodes and nodelets, while the edges correspond the topics used for inter-communication among the nodes. The nodes were colored for a better visual categorization. The red nodes correspond to drivers and low-level nodes (described in section 7.1.1). The base platform node (*pioneer*), the laser node (*sicklms*), Kinect's nodes and nodelets (*openni_node*, *openni_manager*, *kinect_wide_scan* and *pointcloud_throttle*), the Arduino node (*higgs_arduino*) and finally the EKF node (*pose_ekf*) are shown. The green nodes correspond to World and Robot Model nodes. As it can be observed, they all publish relevant transforms to the system, except the *map_server* which publishes the map. We also can observe the navigation component of our system (*move_base_node*) as the blue node, the localization component (*amcl*) as the yellow node, and the mission manager as the purple node.

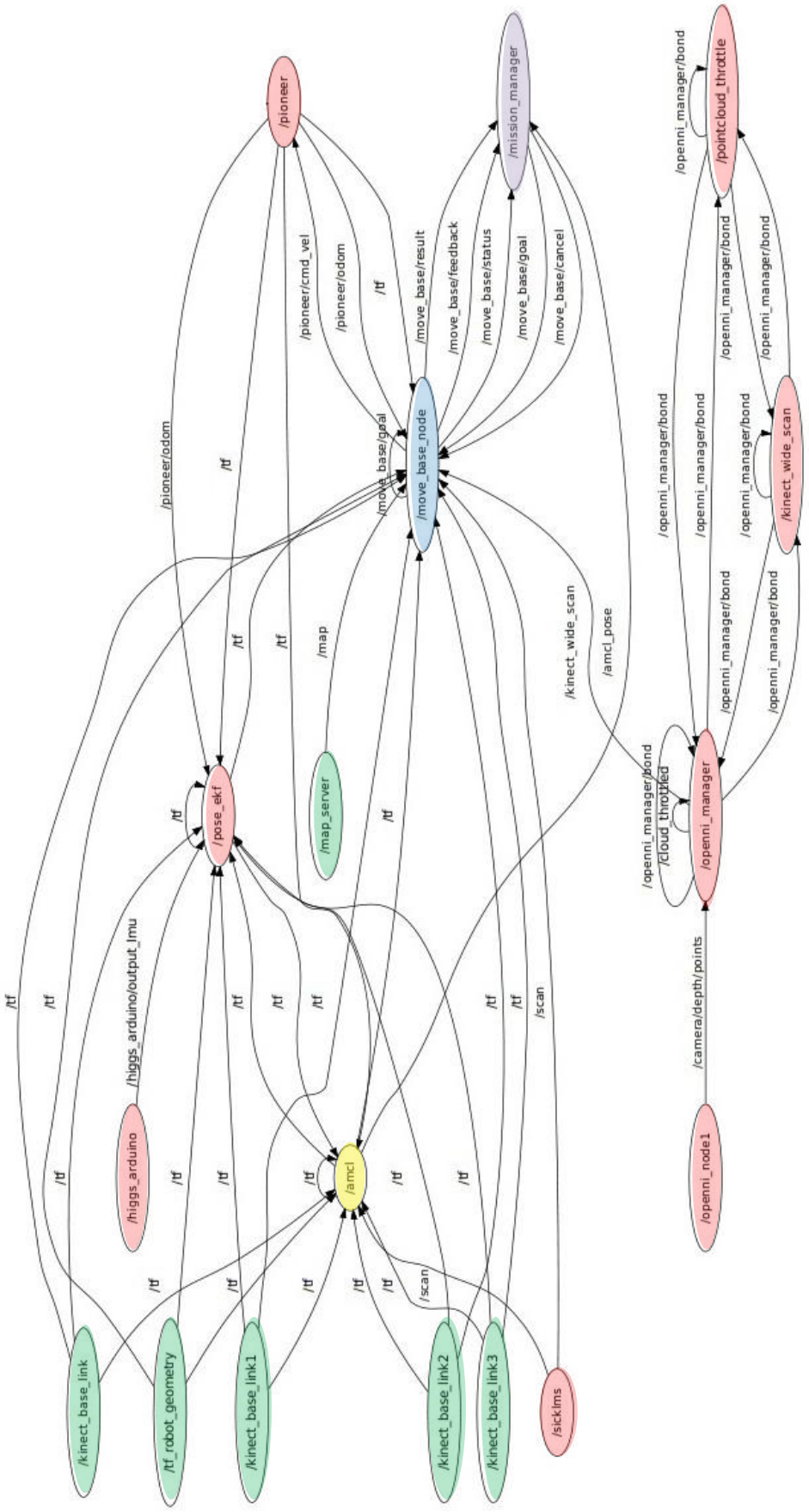


Figure 7.13: The implemented Higgs's base control system, without the meta-control layer

The Table 1 contains a list of all the ROS nodes in the base control system, identified by name, the architecture component they realize and whether they have been reused and properly configured or fully developed for this work.

Name of the ROS node	Functional Use	Reused/Implemented
pioneer	Driver for the base platform	Implemented
sicklms	Driver for the laser device	Reused
openni_node	Driver for the Kinect sensor	Reused
Nodelets: *openni_manager, pointcloud_throttle kinect_narrow_scan, kinect_wide_scan*	Driver for the Kinect, to provide *LaserScan*	Reused
higgs_arduino	ROS wrapper for CORBA driver of Higgs's Arduino board	Implemented
pose_ekf	EKF for pose estimation, through integration of odometry and compass readings	Reused
kinect_base_link, tf_robot_geometry, kinect_base_link1, kinect_base_link2, kinect_base_link3	Corresponds to the robot's model – relevant transforms and robot's coordinate frames. They are published by the transform broadcaster	Implemented
map_server	Corresponds to the World Model. Publishes the map to the system	Reused
amcl	Localization Node	Reused
move_base_node	Navigation Node	Reused
mission_manager	Mission Manager node	Implemented

Table 1: Summary of the reused and implemented nodes for the base control system

7.2 Meta-control Layer Implementation

As with the base control system, the meta-control implementation for Higgs's control system follows its respective architecture described in the chapter 6 - Higgs's Control Architecture. Therefore, a ROS package was developed, named *meta_controller*, which contains an implementation of the three elements of the meta-control layer – the meta-monitor, the meta-model and the meta-actuator. The meta-monitor and the meta-actuator are implemented as two ROS nodes, while the meta-model is defined by a ROS message type, which we called meta-message. The meta-message is published by the meta-monitor node to a ROS topic called */meta_events*, which is received by the meta-actuator (see Figure 7.14). The meta-message contains the name of the event that was detected by the meta-monitor, and information regarding the configuration of the system that needs to be restored by the meta-actuator after the event handling. In our system, this information correspond the current goal position that the *move_base* node is currently trying to achieve.

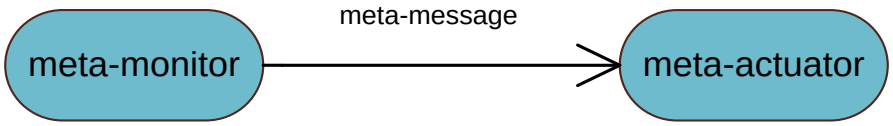


Figure 7.14: Elements of the implemented meta-control layer

In Higgs's particular mission, specified in chapter 5 - Higgs's Mission, a system's reconfiguration is only needed as a reaction to the laser fail event. Therefore, for the implemented system, the meta-model will only include the meta-event "laser fail" that must be detected by the meta-monitor and be handled by the meta-actuator, which needs to reconfigure the system to continue with Higgs's surveillance mission through the *Sala de Calculo*.

Both the meta-monitor and the meta-actuator were implemented using *rospy* (python client libraries for ROS), which was already mentioned in section 4.2.3 - ROS Client libraries and Nodelets. They will be described in the next subsections.

7.2.1 Meta-monitor Node

As already mentioned in the section 6.4 - Meta-control Architecture, the meta-monitor's task is to monitor the system to detect pre-determined events that need to be handled by the meta-actuator. The implemented meta-monitor node monitors the other ROS nodes in the control system by subscribing to the topic */rosout*. The */rosout* topic consists in the standard ROS topic for publishing logging messages. Since most of the ROS nodes in the control system reports error and warnings using the ROS system-wide logging mechanism, the meta-monitor can easily perceive other node's errors by subscribing to the */rosout* topic. This corresponds to the runtime information about components' state that is needed by the designed meta-control. The meta-monitor also uses the ROS command-utility "rosnode list" to know which are the active nodes in the system, with a

pre-set frequency. In our system, this frequency set to 5 fps. Therefore, the meta-monitor can take as long as 0.2 seconds to perceive that a specific node is not active. In our implemented system, the meta-monitor is also subscribed to the topic where the current goal position is published. This way, it can save internally this data, which is later needed for the system re-activation executed by the meta-actuator.

By analyzing the incoming messages from the */rosout* topic and the output of the command-line utility "rosnode list", meta-events can be detected. When this happens, the meta-monitor publishes a meta message to the */meta_events* topic. In our system, this meta-message contains the name of the event (for example, laser fail) and the goal position which Higgs was currently trying to achieve. This latter corresponds to the system configuration needed to be restored when the system is re-activated by the meta-actuator.

7.2.2 Meta-actuator Node

The meta-actuator's main responsibility is to handle the meta-events that are detected by the meta-monitor. In the implemented system, the meta-actuator node is informed of the meta-events by subscribing to the */meta_events* topics, in which it receives meta-messages. When a meta-message is received, its execution steps follow the ones specified in the activity diagram of Figure 6.4:

- ***Interruption of the system:*** the meta-actuator node halts the system. In our implemented system, this is done by canceling the current goal to the *move_base* node, using the *actionlib* standardized ROS interface. This stops Higgs's movement and must be done to prevent error propagation, especially in the localization.

- ***Handling of the event:*** the meta-actuator node reconfigures the system accordingly to the name of the event received in the meta message. This can be done by removing or inserting ROS nodes into the system, as well as changing the parameter's values of the system's nodes. To insert ROS nodes into the system, the meta-actuator uses the ROS command-line utility "rosrun" to run executables in an arbitrary ROS package, or "roslaunch" that launches multiple ROS nodes locally or remotely via SSH. To remove ROS nodes from the system, the meta-actuator uses the ROS command-line utility "rosnode kill" that kills a running system's node, which name is passed as an argument of the command. To set other nodes' parameters' values, the meta-actuator interacts with the ROS's Parameter Server (already mentioned in section 4.2.2 - ROS Concepts). In our system, the meta-actuator handles events using if-else rules. Since the only meta-event corresponds to a laser fail, the implemented meta-actuator analyses the name of the meta-event, and if it corresponds to the "laser fail" event, it will execute the laser fail event handling. The handling of the laser fail event reconfigures the system by removing the laser ROS node (*sicklms*) from the system and inserting the Kinect nodelet (*kinect_narrow_scan*) that will now provide the *LaserScan* needed by the navigation (*move_base*) and the localization (*amcl*) nodes.

- ***Re-activation of the system:*** after the handling of the event, the meta-actuator reactivates the systems, by loading the previously saved system's configuration. In our system, this is done by re-setting the current goal (which was received in the meta-message) to the *move_base* node, using the *actionlib* standardized ROS interface. This resumes Higgs's navigation mission.

7.3 Integration of the Meta-control Layer in the System

The insertion of the meta-control layer in our system is done by simply launching the ROS nodes meta-actuator and meta-monitor, whose implementation was described in the previous section 7.2 - Meta-control Layer Implementation. The meta-monitor node will automatically start to check for error logs, by subscribing to the */rosout* topic and checking for the active nodes in the system, using the command-line utility "rosnode list". Meanwhile, the meta-actuator node subscribes to the */meta_events* topic waiting for meta-messages to be advertised by the meta-monitor..

Once fully implemented the meta-control layer's nodes, the process of their integration into the base system is very simple and transparent due to the modular architecture of ROS, which divides the system in modular executables (ROS nodes). Figure 7.15 shows a cropped part of Higgs's system, obtained by the ROS command-line utility *rxgraph*, where it shows the meta-actuator and the meta-monitor nodes inserted in the system.

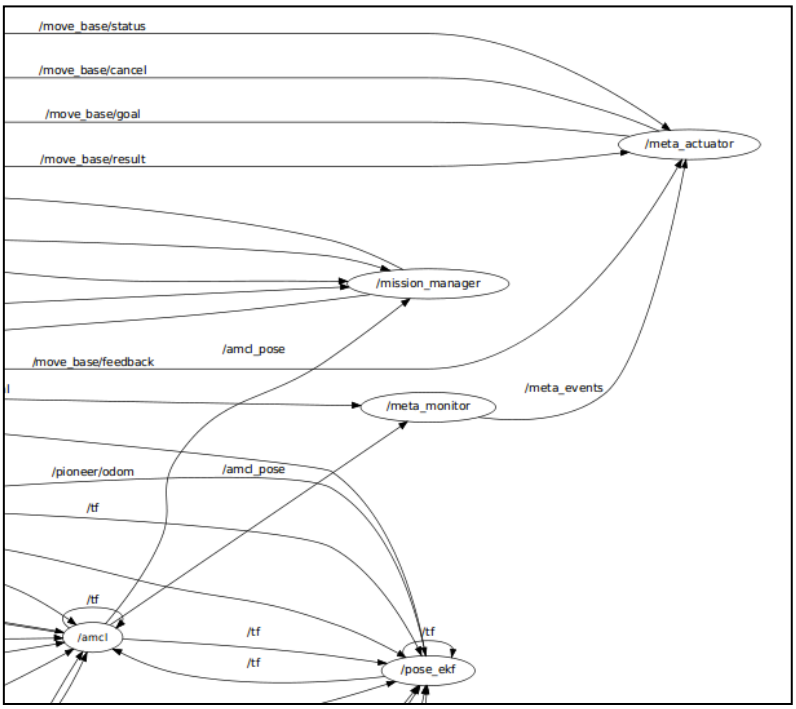


Figure 7.15: Meta-monitor and meta-actuator nodes' insertion in Higgs's control system

However, before inserting these two nodes into Higgs's system, the meta-monitor must know the error/warning that are logged and sent to the */rosout* topic when the laser fails,

in order to properly detect this meta-event. It must also check if any of the active nodes suddenly stops its execution. This information was easily obtained by simulating a laser fail while the control system was running, and checking which errors/warns were sent to the */rosout* topic and which ROS nodes were killed.

8 Experiments and Analysis of the Results

In this chapter, the experiments carried out to validate this work are described and the relevant results are presented and analyzed.

Firstly, the experiments associated with map building are described and some of their results are presented. These experiments were necessary to obtain the map that would be used in Higgs's mission. They were also useful to configure and prepare some of the system's nodes that would later be used for Higgs's mission.

In the second section 8.2, the experiments results associated with Higgs's mission are presented. The efficiency of the base control system is analyzed, as well as the full control system with the meta-control layer.

The third section 8.3 describes some other experiments made with Higgs and its control system, in which the meta-control layer was upgraded to intelligently integrate the Kinect and the laser sensor, while Higgs executes its surveillance mission through the rectangular corridor of *Sala de Calculo*.

8.1 Map Building (Higgs's SLAM system)

To obtain the map of the *Sala de Calculo*, i.e. the map's files needed by the *map_server* node to publish the map to the system, we used the SLAM technique using the laser scan while teleoperating Higgs. The SLAM technique was already mentioned and described in the section 2.4.1 - SLAM.

The SLAM system was implemented in ROS, in which we reused many of the driver and low level nodes implemented for Higgs's control system for its mission. In addition, we added the following ROS nodes:

- *slam_mapping* node – this node's implementation can be found in the *gmapping* package of the ROS repositories. It is a ROS wrapper of the GMapping algorithm developed by Giorgio Grisetti, Cyrill Stachniss and Wolfram Burgard, and described in (Grisetti, Stachniss and Burgard 2006).

- *teleop_base* node – this node's implementation can be found in the *teleop_base* package, available in the ROS repositories. It was implemented by Morgan Quigley and Brian Gerkey, and it is used to teleoperate a robot from a keyboard or a joystick.

For monitoring the map that was being generated and the overall performance of Higgs, we used the ROS tool *rviz*. It was developed by Josh Faust and Dave Hershberger, and

consists in a 3d visualization environment for robot's data using ROS. It enables the visualization of coordinate frames in the system, robot's footprint, the map that is published by the *map_server* node, the laser scan and point scan, and even the particle filter, by subscribing to the proper topics in which these data are published. This tool was also used in the latter described experiments to monitor the control system while Higgs was executing its mission.

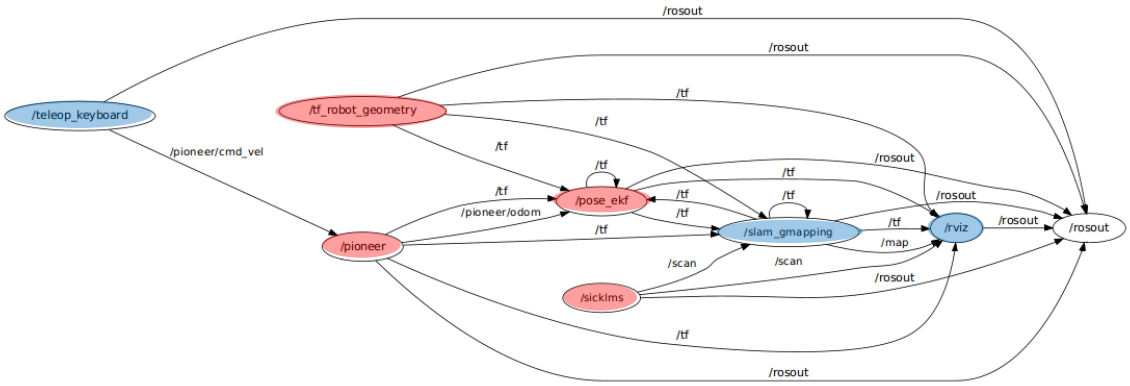


Figure 8.1: ROS node system used for Higgs's SLAM

Figure 8.1 presents the implemented SLAM system used for building the map. This graph was generated using the ROS command-line utility, *rxgraph*, that visualizes the ROS computation graph. The red colored nodes correspond to low-level nodes that are also used in Higgs's control system for the main mission (see chapter 7 - Implementation). These nodes were launched in Higgs's onboard laptop, which was directly connected with the hardware devices used for this control system. The other ROS nodes, colored in blue, were launched in a desktop that was connected to the onboard laptop, through the ASLab's wireless network. This was essential to reduce the computational cost in the onboard laptop, whose computational capabilities were insufficient to hold the entire SLAM system. Since ROS enables the nodes to have a transparent communication among terminals, as long as they are connected to the same network and use the same ROS core, there were no difficulties to launch the nodes of the control system in different terminals.

For the construction of the map, we teleoperated Higgs through the *Sala de Calculo*, while running the SLAM system and monitoring the whole system's performance on *rviz*. After the full construction of the map, we used the command-line utility of *map_server* package - *map_saver* – to save the generated map to the respective files that will be loaded by the *map_server* node in Higgs's mission.

The process of the map generation and SLAM can be seen in Figure 8.2, from the step (1) through the step (8). As you can see, the developed SLAM system successfully built a suitable occupancy grid map of the *Sala de Calculo*, maintaining most of its geometric properties, such as the main rectangular corridor in which Higgs will navigate in its mission. The GMapping SLAM algorithm, used by this system, seemed the overcome the loop closing problem[16]. This can be verified by analyzing the execution steps (7)

[16] Loop closing problem exists when a robot, while mapping an environment, must determine whether or not it is the first time it visits a certain location.

and (8) in Figure 8.2, in which the robot correctly indentifies the spot where it initiated its trajectory. The final map, which was used in Higgs's mission, is shown in Figure 8.3.

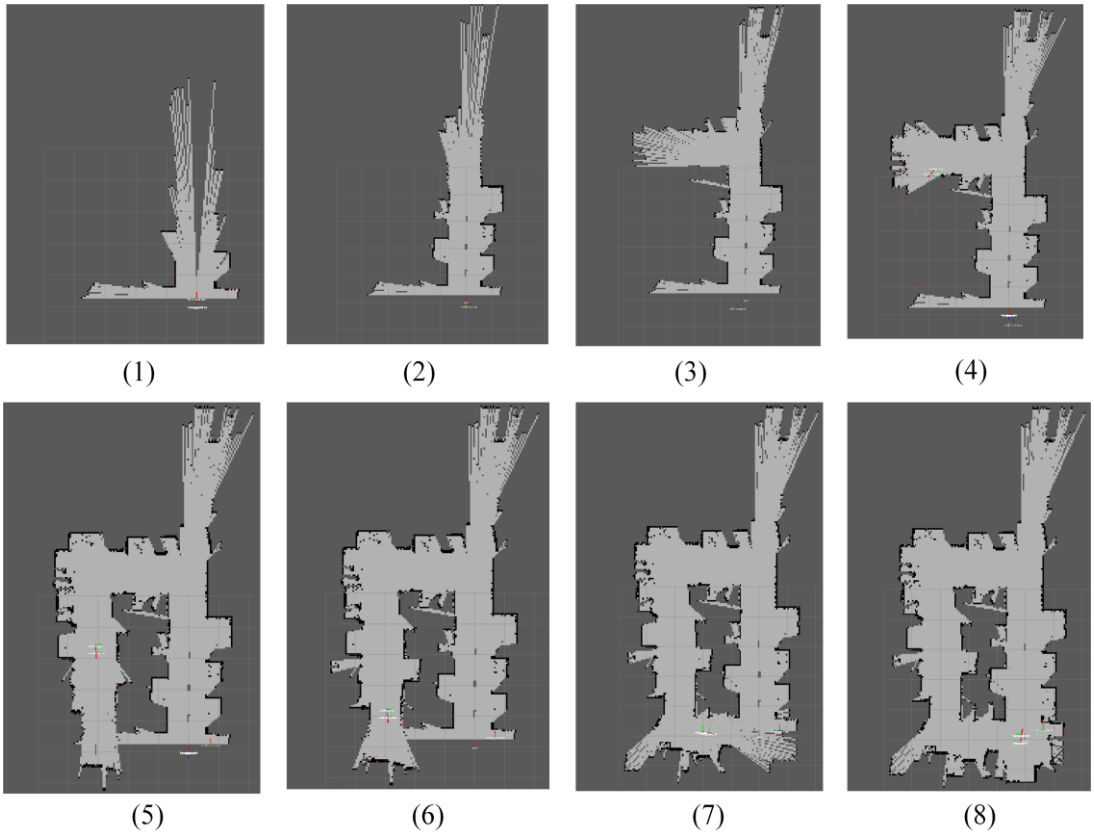


Figure 8.2: Map generation process in Higgs's SLAM system, using the GMapping algorithm

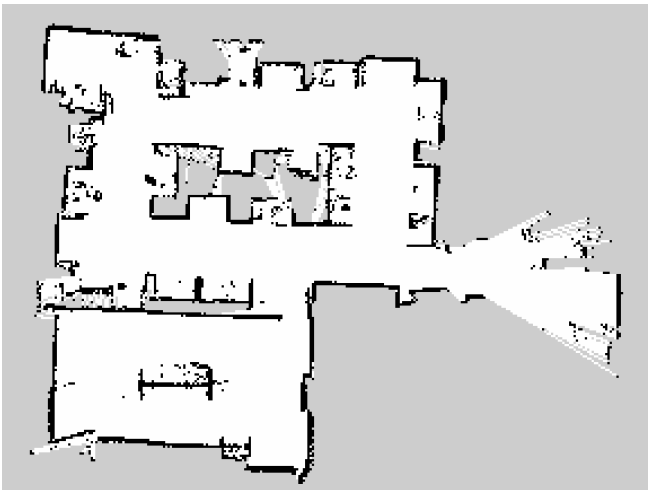


Figure 8.3: Map of *Sala de Calculo* generated by the Higgs's SLAM system

8.2 Higgs's Control System and Higgs's Mission

In this section, the experiments associated with the implemented Higgs's control system for its mission are going to be described, and the results are going to be presented and analyzed. As with the SLAM control system experiments, all drivers and low-level nodes were launched in the onboard laptop, while the remaining nodes were launched in a desktop connected to the laptop through the ASLab wireless network. This was essential to reduce the computational cost in the onboard laptop, whose computational capabilities were insufficient to hold the entire Higgs's control system.

The first section, 8.2.1, refers to the experiments associated with the base control system. These experiments were essential, because it is important to assure that Higgs's base control system is able of autonomously navigating Higgs through the rectangular corridor of *Sala de Calculo* when there are no failures in the system during the execution of the mission. The meta-control layer is used, in Higgs's mission, to replace the sensor used in the system to get the point scan. Therefore, before starting the mission, the base control system must be properly prepared to navigate through the *Sala de Calculo* using both point scan sensor devices – the Kinect and the laser. These experiments will also provide us results to compare the efficiency of the system when using one of these sensor devices.

Experimental results obtained using Higgs's full control system, with the meta-control layer, are going to be presented in the section 8.2.2. These experiments also correspond to Higgs's mission, as it is defined in chapter 5, in which Higgs navigates through the rectangular corridor of *Sala de Calculo*, and recovers from a laser fail if necessary. The results obtained in this section are compared to the ones obtained in the section 8.2.1, to analyze the efficiency of the meta-control layer, and to conclude on its importance to the system when it is vulnerable to a the laser fail.

8.2.1 Higgs's Base Control System

Once having the map of *Sala de Calculo*, the first experiments using Higgs's base control system were made to analyze the efficiency of the localization and the navigation in Higgs. The implemented ROS nodes responsible for these two tasks are the *amcl* node and *move_base* node respectively, although they depend on the other implemented ROS nodes (such as the drivers and the low-level nodes). Therefore, to make these experiments, all ROS nodes in the control system, except the Mission Manager, were launched. We manually set some goal positions, using the *rviz* tool[17], and analyzed the performance of the Higgs's autonomous localization and navigation, while it tried to arrive at the designated goal positions. Some of these results can be

[17] *rviz* consists in the 3d visualization environment for robots using ROS. It enables the visualization of coordinate frames in the system, robot's footprint, the map that is published by the *map_server* node, the laser scan and point scan, and even the particle filter, by subscribing to the proper topics in which these data are published. This tool was also used in Higgs's SLAM system described in the section 8.1.

appreciated in Figure 8.4, in which the laser was used as the point scan sensor. The small red arrows represents the set of particles that *amcl* uses to compute the location of the robot. The manually set goal position is represented by the big red arrow, and the motion path generated by the *move_base* global planner can be seen in green. As you can, see, the robot (represented by its blue footprint) successfully generated a plan, avoided the obstacles and navigated to the designated goal position, achieving it in 24.3 seconds. At the same time, the set of particles converged, as the robot moved, which means that the pose estimation's certainty, computed by the *amcl* node, increased significantly all through the robot's navigation. These results show that the Higgs is capable of autonomous navigation through the *Sala de Calculo*, and that the *amcl* node, the *move_base* node and all the remaining nodes in base control system were properly configured according to Higgs's characteristics.

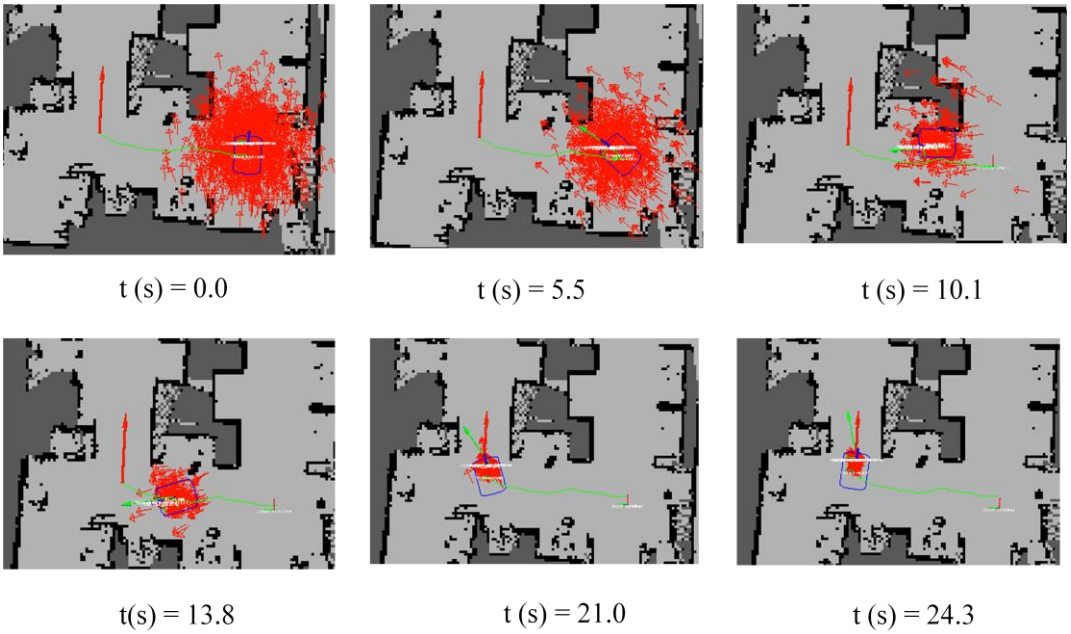
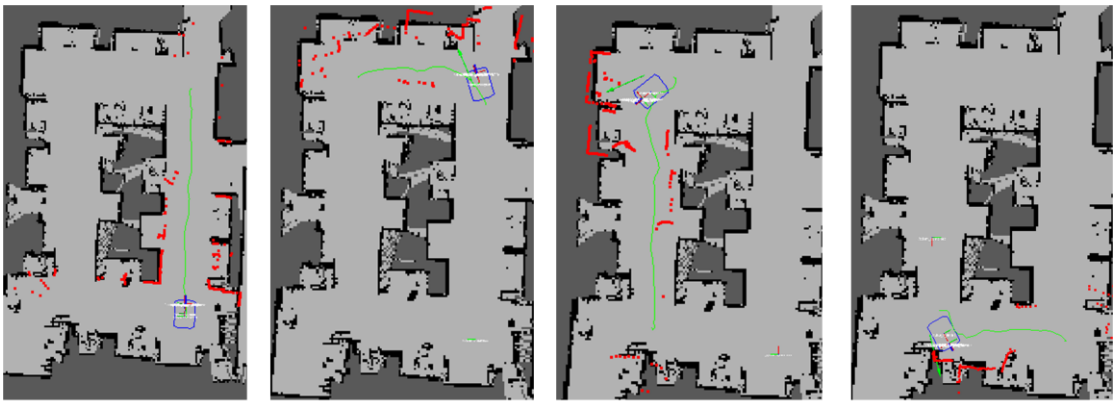


Figure 8.4: The performance of the *amcl* node and the *move_base* node in simple navigation tasks

The next experiments involved the whole base control system, in which the *mission_manager* node was inserted. Experimental observation indicated that Higgs could successfully and autonomously navigate through the rectangular corridor of *Sala de Calculo*, while avoiding obstacles and localizing itself. Some of the navigation results can be seen in Figure 8.5, which shows the motion paths (in green) to the four waypoints that define the close-loop route through the rectangular corridor.



(1) (2) (3) (4)

Figure 8.5: Navigation to the four waypoints specified by the Mission Manager.

Measures were taken as for the time it took for Higgs to make a complete lap through the rectangular corridor of *Sala de Calculo*. We first measured the amount of time that Higgs took to make a full lap through the rectangular corridor, in which the base control system used the laser as the point scan sensor. We then measured the time using the base control system configured to use the Kinect as the point scan sensor. For each of these two devices, seven trials were made and seven time measures were obtained. The results can be seen below, in Table 2, which presents the time measures in seconds.

	Lap 1	Lap 2	Lap 3	Lap 4	Lap 5	Lap 6	Lap 7	Mean ± St Dev
Laser	180.5	198.5	158.9	202.0	153.0	211.6	135.9	177.2 ± 26.4
Kinect	224.9	302.0	346.2	536.4	489.5	331.7	422.2	379.0 ± 101.3

Table 2: Time measures, in seconds, of Higgs's navigation through the rectangular corridors using the laser and the Kinect exclusively

As you can observe in Table 2, using the laser as the main point scan sensor in our system provides better results. This is reflected by the mean time of each lap using the laser – 177.2 seconds – against the mean time of each lap using the Kinect – 379.0 seconds. This means that Higgs navigation is approximately twice as fast when it uses the laser instead of the Kinect as the point scan sensor. Another aspect to notice in the Table 2 is the standard deviation associated with each sensor. The laser's time measures present a relatively low standard deviation, meaning that the results of the time measures are not very different from each other, and the time Higgs takes to make a lap is somewhat predictable. However, the standard deviation associated with the Kinect's time measures is very high (more than 101 seconds), which indicates that the Higgs's control system is more unpredictable as for the time Higgs takes to complete a lap through the rectangular corridor. This latter fact reinforces the idea that using the laser as the point scan sensor, improves the efficiency and the reliability of the control system in Higgs. This discrepancy in the efficiency can be explained by the difference in the horizontal sensing angular ranges of the laser (180º) and Kinect (57º) . The fact that the laser has a wider sensing ranges implies that the *amcl* node can compute a better pose

estimation because it can match more point scans with the map's landmarks. It also helps the *move_base* to detect more obstacles and reduces Higgs's movements to discover lateral obstacles. Therefore, by using the Kinect sensor instead of the laser, Higgs have to drive more carefully because it is less *aware* of its surrounding, and this reflects on the time it takes to make a full lap of the rectangular corridor of *Sala de Calculo*. In addition, it was observed that, while using the Kinect as the point scan sensor, the recovery behaviors (in place rotation) occurred more often. These can be triggered by the *move_base* node when the *amcl* node does not provide a sufficiently precise pose estimation.

8.2.2 Higgs's Full Control System and Higgs's Mission

Before launching the meta-control nodes into the system, we analyzed which errors were logged by the system's nodes when a laser fail occurs. These errors must be detected by the meta-monitor node, and are needed to trigger the reconfiguration in Higgs. Experimental trials indicated that the errors which are published to the */rosout* topic, when we shutdown the laser, are:

- *"Connectivity Error: Could not find a common time /base_link and /map"*: these error is sent by the *move_base* node when the *amcl* node stops publishing the transformation which indicates the robot's pose estimation. The *amcl* cannot publish this transformation because it lacks of laser scan data.

- *"woah! error!"*: this error is sent by the *sicklms* node (which consists of the node driver of the laser) just before it stops its execution due to the laser device shutdown.

Therefore, the meta-monitor node was programmed to detect the "laser fail" event when one of these errors is logged and published to the */rosout* topic, to which the meta-monitor is subscribed. After this, the implemented Higgs's full control system was ready to be launched, and observations could be made on the execution of Higgs's mission when a laser fail is simulated during the mission's execution. To launch Higgs's full control system, it is only needed to launch the base control system as well as the meta-monitor and the meta-actuator node.

The laser fail was simulated by remotely killing the *sicklms* node in some trials, and in others by shutting down Higgs's laser device using the respective power switch of Higgs's power board.

The experimental trials that were made indicated that Higgs was able to recover from the laser fail (which was simulated at different points in Higgs's trajectory) and continue with the execution of its surveillance mission. The laser fail was properly detected by the meta-monitor node, which immediately alerted the meta-actuator. The meta-actuator successfully reconfigured the control system, by removing the non-operational laser ROS node (*sicklms*) from the system and inserting the Kinect nodelet

(*kinect_narrow_scan*) that now provides the *LaserScan* for the navigation (*move_base*) and the localization (*amcl*) nodes. These latter nodes accepted the new *LaserScan* data provided by the Kinect, and continue with their respective operations. It was also observed that immediately after the laser fail was simulated, Higgs's halted its movement (due to the system's interruption by the meta-actuator by cancelling the current goal), and, after a few seconds, it resumed its navigation to the same goal position it had previously (due to the system's re-activation made by the meta-actuator, in which it re-sets the goal position provided by the meta-monitor).

Figure 8.6 shows two Higgs's system state at two different moments in time of the same trial[18]. In Figure 8.6 (a), it can be observed that Higgs is using the laser as the point scanner device, since the scanned points (in red) cover a horizontal range of approximately 180 degrees. The visualization represented by Figure 8.6 (b) was obtained after a laser fail, which occurred at the point of trajectory indicated by the yellow cross in Figure 8.6 (a). As it can be seen, the Higgs's system reconfigured itself to use the Kinect as the point scanner device, since the scanned point cover a much smaller angular range (of approximately 60 degrees), and Higgs continued its trajectory towards the same goal position, while localizing itself.

(a) (b)

Figure 8.6: Higgs navigation before (a) and after (b) the laser fail.

[18] This images were obtained using the ROS visualization tool - *rviz*.

	Lap 1	Lap 2	Lap 3	Lap 4	Lap 5	Lap 6	Lap 7	Mean ± St Dev
Lap Time	208.8	242.9	238.3	214.5	276.7	367.0	245.0	256.2 ± 49.7
Reconfiguration Time	12.2	11.1	12.4	11.7	11.0	10.7	10.6	11.4 ± 0.7

Table 3: Time measures, in seconds, of Higgs's navigation through the rectangular corridors with a simulated laser fail

Table 3 shows the amount of time, in seconds, it takes Higgs to make a full lap through the rectangular corridor of *Sala de Calculo*, when a laser fail occurs. As with the base control system, we made seven trials and calculated the mean and the standard deviation. The system's reconfiguration time was also measured. The laser fail was simulated always at the same point in the trajectory, which corresponds to the one indicated by the yellow cross in Figure 8.6 (a). Comparing this results with the Table 2, you can conclude that the mean value is slightly higher to the one measured for a navigation with the laser (256.2 seconds vs. 177.2 seconds, respectively). This means that the laser fail event only increased 79 seconds to the amount of time it takes Higgs to make a full lap through the rectangular corridor, compared to a situation in which the laser wouldn't have fail. This can be considered a good result, since the efficiency of the mission have not decreased significantly and Higgs was able to continue its navigation mission, despite navigating a little slower. The decrease in the navigation speed can be explained by the fact that, after the reconfiguration, Higgs uses the Kinect sensor, which, as already mentioned in the previous section, proved to make the control system less efficient due to its reduced angular field of view (57 degrees) compared to the laser's field of view (180 degrees).

Another measure to take into account, presented in the Table 3, is the Reconfiguration time, whose mean value is approximately 11 seconds. Therefore, after a failure, Higgs takes only 11 seconds to reconfigure itself, and to re-activate the system. This means that, if we had in our system an equally efficient alternative to the laser device that would be used to provide point scans after the reconfiguration, the increment in the amount time to complete a lap will only be of approximately 11 seconds. The standard deviation of measured values of reconfiguration times, which is very low, indicates that this time is very constant and predictable.

To conclude, analyzing the experimental results, we can affirm that Higgs, along with the implemented control system, is capable of achieving all goals for its surveillance mission, which were specified in chapter 5 - Higgs's Mission.

8.3 Other Experiments

In this section some further experiments are going to be described, in which the meta-control layer was upgraded to intelligently integrate both point scan sensors – the Kinect and the laser, constituting a hybrid system.

Since the Kinect power consumption (12W) is far inferior to the laser's device power consumption (40W), it is suitable to use the Kinect whenever it does not affects significantly the efficiency of the mission. Therefore, for these experiments, we set the Kinect as the default point scan sensor of our system, and re-programmed the meta-control layer to reconfigure the system to use the laser when Higgs encounters difficulties in computing the pose estimation. The laser would be used until the current goal position is reached. After that, the meta-control layer reconfigures the system so that the Kinect would be used again as the solely provider of point scan to the system. Thus, there are two basic states in our system: one in which the Kinect is used as the point scan sensor; and another in which the laser is used as the point scan sensor. The commutation of these states can occur several times during the navigation. This way, the Kinect is used every time Higgs localizes itself with precision, and the laser is used when Higgs feels a little lost and needs to localize itself rapidly.

Previous experimental observations proved that the following error was logged to the */rosout* topic when *amcl* node had difficulties in computing the pose estimation: "*Costmap2DROS transform timeout*". This error is sent by the *move_base* node when the pose estimation is not published into the system at the desired frequency. This error was used to detect when Higgs encounter difficulties in computing its pose estimation while using the Kinect sensor.

The meta-monitor was re-programmed to detect this error and to inform the meta-actuator of the "Pose estimation alert" event. When receiving the meta-message associated with the "Pose estimation alert", the meta-actuator reconfigures the system by removing the Kinect nodelet (*kinect_narrow_scan*) and inserting the laser node (*sicklms*) that would be used to provide point scan to the system. While using the laser, the meta-monitor will wait for Higgs to arrive to the current goal position, and when this happens, it will send another meta-message to the meta-actuator, signaling the "Arrival at the goal position while using laser" event. The meta-actuator handles this event the same way it handles a laser fail: it removes the laser node (*sicklms*) from the system and inserts the Kinect nodelet (*kinect_narrow_scan*) that will provide the *LaserScan* for the navigation (*move_base*) and the localization (*amcl*) nodes.

	Trial 1	Trial 2	Trial 3	Trial 4	Trial 5	Trial 6	Trial 7	Mean (Mean %)
Kinect time	98.0 (37.0%)	67.0 (27 %)	93.0 (40 %)	105.9 (39.2 %)	132.9 (42.3 %)	132.5 (50.4 %)	148.3 (57.1 %)	111.1 (42.0 %)
Laser time	126.0 (48.0 %)	134.0 (58 %)	108.0 (45 %)	123.2 (45.7 %)	140.0 (44.6 %)	88.9 (33.9 %)	69.7 (26.9 %)	112.8 (43.1 %)
Reconfiguration Time	40.0 (15.0 %)	33.0 (15 %)	37.0 (15 %)	40.8 (15.1 %)	41.3 (13.1 %)	41.1 (15.7 %)	41.4 (15.9 %)	39.2 (14.9%)
Total Time	264	234	238	269.9	314.2	262.5	259.4	263.1 (100.0%)

Table 4: Time measures, in seconds, of Higgs's navigation through the rectangular corridors, using the hybrid system of Kinect and Laser

Table 4 shows the amount of time, in seconds, it takes Higgs to make a full lap through the rectangular corridor of *Sala de Calculo*, using this hybrid system. As with the base control system, we made seven trials and calculated the mean and the standard deviation. The use percentage of each sensor for each lap corresponds to the values in parenthesis. As it can be seen by the values on the table, this system uses the Kinect and the Laser sensor in approximately equally distributed shares of time. This can be verified by their mean time of usage values, which are very similar (111.1 seconds vs. 112.8 seconds). In each of the seven trial, Higgs's self-reconfigured itself four times. Almost 15% of the time was spent in reconfigurations, which is a relatively low percentage. The number of reconfigurations can be reduced by increasing the tolerance of the desired frequency in which the pose estimation must be publish to the system, which decreases the probability of the error that triggers the "Pose estimation alert" event.

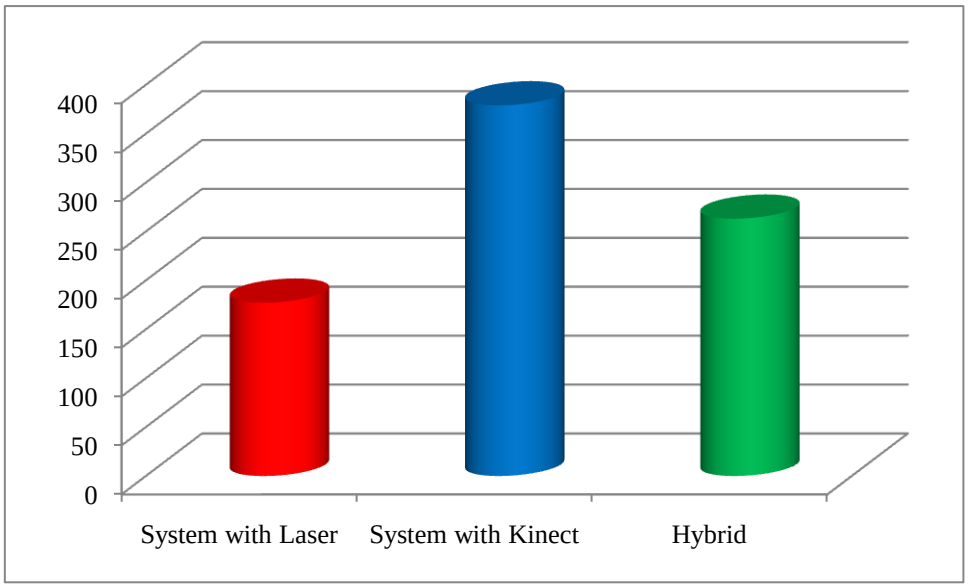


Figure 8.7: Comparison of the mean values for a lap time using a system with laser, Kinect and a hybrid system

The Figure 8.7 shows that this hybrid system corresponds to an alternative robot control system that reduces the power consumption of a robot that uses the laser exclusively and, at the same time, reduces the navigation time of a robot that uses the Kinect

exclusively. It is important to mention that this hybrid system was developed without any changes to the base control system. Only the meta-control nodes were reprogrammed (which followed the meta-control architecture), preserving the remaining system's nodes. Therefore, the intelligent integration of this two sensors was made possible due to the meta-control layer, which was attached to the base control system.

9 Conclusions

In this document, the implementation of an architecture for robust autonomy is presented. As a result, a robot control system, capable of autonomous self-reconfiguration, was implemented. This enabled an autonomous mobile robot to handle device failures while executing a surveillance mission. The implemented control system is not intended to be exclusively fault-tolerant, but somehow, self-aware. The self-awareness level in this system is not pretended to be comparable to the one that exists in human beings or even in natural systems in general. However, the experimental results show that this system is able to detect both internal errors and mission circumstances and intelligently react to it, which can be considered as a step (though very initial) towards self-awareness.

The main contribution of this work consists in the developed architecture used to implement the robot control system. This architecture is based on the cognitive architecture *The Operative Mind*, included in the ASys Project of the ASLab group of the Technical University of Madrid (UPM). In this architecture approach, a meta-control layer is inserted to monitor and reconfigure the system whenever necessary. Unlike fault-tolerant architectures, this architecture specifies a system that is not only fault-reactive, but can use system's information to increase its efficiency. To this end, three main elements in the meta-control layer were defined:

- a meta-monitor that monitors the system to detect possible meta-events;

- a meta-actuator that handles the meta-events, by interrupting the system, reconfiguring the system accordingly to the received meta-event, and re-activating the system;

- a meta-model, used by the meta-monitor and the meta-actuator, which defines the type of possible meta-events and the system's configuration that needs to be restored after the meta-event handling.

To test the architectural approach, we specified a simple but challenging indoor surveillance mission to be executed by an autonomous mobile robot, in which a robot's device (laser) fails during the execution of the mission. Therefore, the system needed to reconfigure itself to overcome the fail occurred in the system.

A control system for the autonomous mobile robot was implemented using the ROS middleware robotic platform. This control system is based on the designed architecture, thus containing a meta-control layer, and follows all the requirements of the specified mission. This thesis presents a detailed description of the implemented system, as for the nodes used and their functions. The mobile robot used for the mission was Higgs, a Pioneer 2-AT8 robotic platform that is part of the ASys Robot Control Testbed (RCT). Higgs is equipped with many onboard devices, such as a radial laser and a Microsoft's Kinect sensor.

Experimental results were presented and analyzed in order to show that our system successfully controls the robot to fulfill its mission in all its aspects:

- The robot is able to autonomously navigate through the specified environment, localizing itself and avoiding dynamic and static obstacles.

- The robot is able to autonomously recover from the laser fail that occurs during the mission, and continue the execution of the mission with no further problems.

Therefore, the meta-layer inserted in the control system proved to be essential for the mission, because it is responsible for the system's reconfiguration. In addition, experimental results demonstrated that the interruption and re-activation of our system, before and after its reconfiguration, were crucial to avoid our robot to get lost in the environment. Without these processes and after the system's reconfiguration, the localization module had many difficulties in computing a pose estimation. By measuring navigation time, we could also verify that the robot navigates more rapidly when it uses the laser as the main radial scanner sensor than using the Kinect sensor.

Finally, some further experiments demonstrated that our system can be easily improved by upgrading the meta-control layer while keeping the base control system unchanged. By expanding the meta-model, i.e. the meta-events that can occur in our system, and by programming the meta-monitor to detect these new meta-events and the meta-actuator to handle them, we upgraded out robotic control system to intelligently integrate the robot's main scanner sensors – Kinect and laser. This upgraded system uses the Kinect as the default sensor, which consumes less energy. When the robot has difficulties in computing its pose estimation, the system reconfigures itself to temporarily use the laser, which has higher energy consumption but increases the efficiency of the pose estimation. Thus, this hybrid system is more power efficient that the system that uses exclusively the laser, and the robot's navigation if faster compared to the system that uses exclusively the Kinect.

On the other hand, the meta-operating system ROS proved to be a suitable robotic software platform for the implementation of our system, mainly because of the available monitoring utilities and its large set of implemented algorithms and drivers that we could reuse for our system. This helped us to focus our work in the implementation of the meta-control system. However, a considerable amount of time was spent to adapt the reused ROS components to our robot and to our environment's characteristic.

Furthermore, we can state that the final implemented system is relatively stable and robust. In our society, it is difficult to find robust mobile robots that are capable of autonomously patrolling a partially known area, and it is even more difficult to find one capable of handling failures, or intelligently integrating two redundant devices, such as the laser and the Kinect. Therefore, this work can also be suitable for those who search for a guideline of how implementing a robust control system into an autonomous mobile robot or how to implement a robotic control system that deals with redundant devices.

9.1 Future Works

The implemented control system can be compared to a fault-tolerant robotic control system. However, one of the main goals of this work is to provide a general architectural design and implementation that could be extended in the future in any direction of interest. These extensions could improve even more the robustness of this robotic control system, thus allowing the robot to deal with even more complex,

dynamic and unknown environments. We already verified that the implemented meta-control layer can be easily upgraded while preserving the implementation of the underlying base control system. Therefore, is up to new research works to keep upgrading this architecture or the implemented (meta)control system, to enhance the robustness and autonomy of mobile robots, and even other kinds of technical systems.

The implemented meta-control layer in our system was programmed with simple "*if-else*" rules. Nonetheless, the designed architecture does not specify how the meta-control components have to be implemented. Some more sophisticated Artificial Intelligence techniques can be used to implement these components, such as Markov decision processes, Bayesian networks or Artificial Neural Networks. In addition, machine learning techniques could also be implemented to enable the robot to learn and adapt autonomously from interactions with the environment.

Bibliography

Agre, P., and D. Chapman. "PENGI: An implementation of a theory of activity." *Proceedings of the Sixth National Conference on Artificial Intelligence (AAAI-87).* Seattle, WA, 1987. 268-272.

Albus, J., A. Meystel, A. Barbera, M. Del Giorno, and R. Finkelstein. "4D/RCD: A reference model architecture for unmanned vehicle systems version 2.0." Technical Report, National Institute of Standards and Technology, 2002.

Albus, J.S., and A.J. Barbera. "RCS: A Cognitive Architecture for Intelligent Multi-Agent Systems." *Proceedings of the 5th IFAC/EURON Symposium on Intelligent Autonomous Vehicles, IAV 2004,*. Lisbon, Portugal, 2004.

Ambros-Ingerson, J., and S. Steel. "Integrating planning, execution and monitoring." *Proceedings of the Seventh National Conference on Artificial Intelligence (AAAI-88).* St. Paul, MN, 1988. 83-88.

Anderson, J. R., and K. Gluck. " What role do cognitive architectures play in intelligent tutoring systems?" In *Cognition & Instruction: Twenty-five years of progress*, 227-262. 2001.

Araki, M. "PID Control." In *Control Systems, Robotics, and Automation*. 2000.

Baydar, C.M., and K. Saitou. "Off-line error prediction, diagnosis and recovery using virtual assembly systems." *IEEE International Conference on Robotics and Automation.* 2001. 818-823.

Bermejo-Alonso. "OASys: ontology for autonomous systems." PhD Thesis, E.T.S.I.I.M., Universidad Politécnica de Madrid, 2010.

Blanke, M, M Kinnaert, J. Lunze, and M Staroswiecki. *Diagnosis and Fault-Tolerant Control.* Berlin: Springer-Verlag Berlin Heidelberg, 2006.

Bongard, J.C., and H. Lipson. "Automated Robot Function Recovery after Unanticipated Failure or Environmental Change using a Minimum of Hardware Trials." *Conference on Evolvable Hardware (EH'04).* Seattle, Washington, USA, 2004. 169-177.

Booth, Taylor L. *Sequential Machines and Automata Theory.* New York: John Wiley and Sons, Inc., 1967.

Bratman, M. E. "Plans and resource-bounded practical reasoning." *Computational Intelligence*, 1988: 349-355.

Bratman, M. E., D. J. Israel, and M. E Pollack. "Plans and resource-bounded practical reasoning." *Computational Intelligence*, 1988: 349-355.

Brooks, R. A. "Intelligence without representation." *Artificial Intelligence*, 1991b: 47:139-159.

Brooks, R. A. "Elephants don't play chess." *Designing Autonomous Agents*, 1990: 3-15.

Brooks, R.A. "Intelligence without reason." *Proceedings of the Twelfth International Joint Conference on Artificial Intelligence (IJCAI-91).* Sydney, Australia, 1991a. 569-595.

Brooks, R.A. "A robust layered control system for a mobile robot." *IEEE Journal of Robotics and Automation,* 1986: 2(1):14-23.

Calisi, D, A Censi, L Iocchi, and D Nardi. "OpenRDK: a modular framework for robotic software development." *Proceedings of the International Conference on Intelligent Robots and Systems (IROS).* Nice, France, 2008. 1872--1877.

Chapman, D., and P Agre. "Abstract reasoning as emergent from concrete activity." *Reasoning About Actions &Plans - Proceedings of the 1986 Workshop.* San Mateo, CA: Morgan Kaufmann Publishers, 1986. 411-424.

Cohen, P. R., M. L. Greenberg, D. M. Hart, and A. E. Howe. "Trial by fire: Understanding the design requirements for agents in complex environments." *AI Magazine*, 1989: 32-48.

Conley, Ken. *Introduction to ROS.* 2011. http://www.ros.org/wiki/ROS/Introduction (accessed 2011).

Driankov, D., and A Saffiotti. *Fuzzy Logic Techniques for Autonomous Vehicle Navigation.* Springer-Verlag, 2001.

Egmont-Petersen, M., de Ridder, and Handels, H. D. "Image processing with neural networks - a review." 2279–2301. 2002.

Etzioni, O., N. Lesh, and R. Segal. "Building softbots for UNIX." *Software Agents - Papers from the 1994 Spring Symposium (Technical Report SS-94-03),* 1994: 9-16.

Ferguson, I. A. *TouringMachines: An Architecture for Dynamic, Rational, Mobile Agents.* Clare Hall, University of Cambridge, UK: PhD thesis, 1992.

Fikes, R. E., and N. Nilsson. "STRIPS: A new approach to the application of theorem proving to problem solving." *Artificial Intelligence*, 1971: 189-208.

Fox, D. "Adapting the Sample Size in Particle Filters Through KLD-Sampling." *International Journal of Robotics Research*, 2003.

Fox, D., W. Burgard, and S. Thrun. "The dynamic window approach to collision avoidance." *Robotics & Automation Magazine*, 1997: 23–33.

Garcia, M. *Fusión sensorial en plataforma robótica móvil (Sensing fusion in a robotic mobile platform).* 2009.

Genesereth, M. R, and N. Nilsson. *Logical Foundations of Artificial Intelligence.* San Mateo, CA: Morgan Kaufmann Publishers, 1987.

Georgeff, M. P., and A. L Lansky. "Reactive reasoning and planning." *Proceedings of the Sixth National Conference on Artificial Intelligence (AAAI-87).* Seattle, WA, 1987. 677-682.

Georgeff, M. P., and F. F. Ingrand. "Real-time reasoning: the monitoring and control of spacecraft systems." *Proceedings of the sixth conference on Artificial intelligence applications.* 1990. 198–204.

Georgeff, M., D. Kinny, and M. Wooldridge. "The dMARS Architecture: A Specification of the Distributed Multi-Agent Reasoning System." *Journal of Autonomous Agents and Multi-Agent Systems*, 2004: 5-53.

Gini, M., and G. Giuseppina. "Towards automatic error recovery in robot programs." *Proceedings of the Eighth international joint conference on Artificial intelligence - Volume 2.* San Francisco, CA, USA: Morgan Kaufmann Publishers Inc., 1983. 821-823.

GOSTAI. "The Urbi Software Development Kit, version 2.6." Development Kit, 2010.

Grisetti, Giorgio, Cyrill Stachniss, and Wolfram Burgard. "Improved Techniques for Grid Mapping with Rao-Blackwellized Particle Filters." *IEEE Transactions on Robotics*, 2006.

Hart, P. E., N. J. Nilsson, and B Raphael. "A Formal Basis for the Heuristic Determination of Minimum Cost Paths." *IEEE Transactions on Systems Science and Cybernetics.* 1968. 100–107.

Hernandez, C. "Estudio de las arquitecturas cognitivas: La necesidad de incorporar mecanismos de autoconsciencia en los sistemas de control inteligente." MSc Thesis, Madrid, Spain, 2007.

Hernandez, C., I. Lopez, and R. Sanz. "The Operative Mind: a functional, computational and modeling approach to machine consciousness." *International Journal of Machine Consciousness* (World Scientific Publishing Company), 2009: 83-98.

Hertzberg, J., and F. Schönherr. "Concurrency in the DD&P Robot Control Architecture." *Proceedings of The International NAISO Congress on Information Science Innovations.* 2001. 1079-1085.

Isla, D. "Handling Complexity in the Halo 2 AI." *Gamastura online.* March 11, 2005. (accessed December 15, 2010).

Jalote, P. *Fault Tolerance in Distributed Systems.* New Jersey: PTR Prentice Hall, 1994.

Kaelbling, L. P., and S. J Rosenschein. "Action and planning in embedded agents." *Designing Autonomous Agents,* 1990: 35-48.

Laird, J. E., A. Newell, and P. S. Rosenbloom. "Soar : an architecture for general intelligence." *Artificial Intelligence,* 1987: 33.

Lebiere, C., and J. R. Anderson. "A connectionist Implementation of the ACT-R production system." *Proceedings of the Fifteenth Annual Conference of the Cognitive Science Society.* Mahwah, NJ: Lawrence Erlbaum Associates, 1993. 635-640.

Leonard, John J, and Hugh F. Durrant-Whyte. "Mobile Robot Localization by Tracking Geometric Beacons." *IEEE Transactions in Robotics and Automation*, 1991: 376-382.

Lopez, I. "A Fundation for Perception in Autonomous System." PhD Thesis, Madrid, Spain, 2007.

Maes, P. "The agent network architecture (ANA)." In *SIGART Bulletin*, 115-120. 1991.

Makarenko, A., A. Brooks, and T. Kaupp. "Orca: Components for Robotics." *International Conference on Intelligent Robots and Systems (IROS).* 2006.

Markoski, B., S. Vukosavljev, D. Kukolj, and S. Pletl. "Mobile robot control using self-learning neural network." *Intelligent Systems and Informatics, 2009. SISY '09. 7th International Symposium on.* Subotica, 2009. 45 - 48.

Montemerlo, M., S Thrun, D Koller, and B. Wegbreit. "FastSLAM: A factored solution to the simultaneous localization and mapping problem." *Proceedings of the AAAI National Conference on Artificial Intelligence.* Edmonton, 2002.

Müller, J. P. "A conceptual model of agent interaction." *Draft proceedings of the Second International Working Conference on Cooperating Knowledge Based Systems (CKBS-94).* University of Keele, UK., 1994. 389-404.

Munneke, D., K. Wahlstrom, and L. Zaccara. "Intelligent Software Robots on the Internet." *Artificial Intelligence*, 1998: 1-52.

Newell, A., and H. A. Simon. "Computer science as empirical enquiry." In *Communications of the ACM*, 113-126. 1976.

Nilsson, N. "Teleo-Reactive Programs and the Triple-Tower Architecture." *Electronic Transactions on Artificial Intelligence*, 2001: 99-110.

Ozkan, M., G Kirlik, O. Parlaktuna, A. Yufka, and A. Yazici. "A Multi-Robot Control Architecture for Fault-Tolerant Sensor-Based Coverage." *International Journal of Advanced Robotic Systems.* Croatia, 2010. 67-74.

Pokahr, A., L Braubach, and W. Lamersdor. "Jadex: A BDI Reasoning Engine." In *Multi-Agent Programming*, by R Bordini, M Dastani, J Dix and A Seghrouchni, 149-174. Springer, 2005.

Pollack, M. E., and M. Ringuette. "Introducing the Tileworld: Experimentally evaluating agent architectures." *Proceedings of the Eighth National Conference on Artificial Intelligence (AAAI-90).* Boston, MA, 1990. 183-189.

Rao, S.A., and M.P. Georgeff. "BDI Agents: From Theory to Practice." *Proceedings of the First International Conference on Multiagent Systems (ICMAS'95).* 1995.

Rosenblatt, J. "DAMN: A Distributed Architecture for Mobile Navigation." Carnegie Mellon University, 1997.

Sacerdoti, E. "Planning in a hierarchy of abstraction spaces." *Artificial Intelligence*, 1974: 115-135.

Sacerdoti, E.D. "The non-linear nature of plans." *Proceedings of the Fourth International Joint Conference on Artificial Intelligence.* Stanford, CA, 1975. 206-214.

Sánchez, José Alberto Arcos. "Sistema de navegación y modelado del entorno para un robot móvil." 2009.

Sanz, R., and J. Zalewski. "Pattern-based control systems engineering." *IEEE Control Systems Magazine*, June 2003: 43-60.

Sanz, R., I. Lopez, M. Rodriguez, and C. Hernandez. "Principles for consciousness in integrated cognitive control." *Neural Networks*, 2007: 938-946.

Secchi, H., V. Mut, R. Carelli, H. Schneebeli, M. Sarcinelli, and T. Freire Bastos. "A hybrid control architecture for mobile robots. Classic control, behaviour based control, and petri nets." The Pennsylvania State University, 1999.

Srinivas, S. "Error recovery in robots through failure reason analysis." *Proceedings of the National Computer Conference.* 1978. 275-282.

Steels, L. "Cooperation between distributed agents through self organization." *Proceedings of the First European Workshop on Modelling Autonomous Agents in Multi-Agent Worlds (MAAMAW-89).* Amsterdam, The Netherlands: Elsevier Science Publishers B.V., 1990. 175-196.

Stone, P., and M. Veloso. "Multiagent Systems: A Survey from a Machine Learning Perspective." In *Autonomous Robots*, 345-383. Kluwer Academic Publishers, 2000.

Thrun, S, W. Burgard, and D. Fox. *Probabilistic Robotics.* Cambridge, MA: MIT Press, 2005.

Vikhorev, K. S., N. Alechina, and B. Logan. "The ARTS Real-Time Agent Architecture." *Proceedings of Second Workshop on Languages, Methodologies and Development Tools for Multi-agent Systems (LADS2009).* Turin, Italy, 2009. Vol-494.

Wise, Melonee. *ROS Tutorials: UnderstandingTopics.* 2011. (accessed 2011).

Wood, S. *Planning and Decision Making in Dynamic Domains.* Chichester, England: Ellis Horwood, 1993.

Wooldridge, M. "Conceptualising and Developing Agents." *Proceedings of the UNICOM Seminar on Agent Software.* London, 1995. 42.

Wooldridge, M., and N. Jennings. "Agent Theories, Architectures, and Languages: A Survey." *Lecture Notes in Computer Science* (Cambridge University Press), 1995: 1 - 39.