Technical University of Crete
Department of Electronic and Computer Engineering

monas
robotic platform

A Flexible Software Architecture for Robotic Agents

Alexandros Paraschos

Thesis Committee:

Assistant Professor Michail G. Lagoudakis (ECE)

Assistant Professor Vasilios Samoladas (ECE)

Dr. Nikolaos Spanoudakis (Department of Sciences)

Chania, December 2010

Πολυτεχνείο Κρήτης
Τμήμα Ηλεκτρονικών Μηχανικών και Μηχανικών Υπολογιστών

monas
robotic platform

Μια Ευέλικτη Αρχιτεκτονική Λογισμικού για Ρομποτικούς Πράκτορες

Αλέξανδρος Παράσχος

Χανιά, Δεκέμβριος 2010

Abstract

In present days, the advancement of robotic technology has opened several possibilities for deployment in several domains, such as search and rescue, surveillance, housekeeping tasks, elderly care, etc. However, despite the improved hardware, the reduced costs, and the increased production rates of robotic systems, the development of such consumer applications is still a domain of experts. This thesis describes Monas, a flexible software architecture for the development of general-purpose robotic agents. Currently, robot software developers have to overcome a variety of problems secondary to their main task of development at the algorithmic and software level, such as hardware differences, constantly evolving robot APIs, cross-compilation issues, performance issues, and the lack of generic third-party software libraries. Monas aims at addressing these problems by providing an abstraction from the robot hardware level and a generic framework for synthesizing robotic agents. In the quest of utilizing principled software engineering methodologies in the context of robotics, we have integrated an Agent Oriented Software Engineering (AOSE) methodology with Monas. Specifically, the Agent Systems Engineering Methodology (ASEME) was used for developing the software for a physical robot team competing in the Standard Platform League of the RoboCup competition (the robot soccer world cup). The team is composed by three humanoid robots who play soccer autonomously in real time utilizing the on-board sensing, processing, and actuating capabilities, while communicating and coordinating with each other in order to achieve their common goal of winning the game. Our work addresses mainly the challenge of coordinating the robot's base functionalities (object recognition, localization, motion skills) in order to present a desired team behavior. Our framework demonstrates the added value of using an AOSE methodology in robotics, as ASEME allowed for compact representations of the state of the agents and complex conditional state transitions, automated a large part of the code generation process, and reduced the total development time for the agents.

Περίληψη

Στην σημερινή εποχή, η εξέλιξη της ρομποτικής τεχνολογίας έχει ανοίξει νέες προοπτικές για ανάπτυξη σε διάφορους τομείς, όπως η αναζήτηση και διάσωση, η επιτήρηση χώρων, οι οικιακές εργασίες, η φροντίδα ηλικιωμένων, κλπ. Ωστόσο, παρά τη βελτίωση του υλικού, το μειωμένο κόστος και την αυξημένη παραγωγή ρομποτικών συστημάτων, η ανάπτυξη τέτοιων εφαρμογών για το ευρύ καταναλωτικό κοινό, εξακολουθεί να απαιτεί την εμπλοκή εμπειρογνωμόνων. Στην εργασία αυτή παρουσιάζεται η Monas, μια ευέλικτη αρχιτεκτονική λογισμικού για ανάπτυξη ρομποτικών πρακτόρων γενικής χρήσης. Επί του παρόντος, οι προγραμματιστές λογισμικού ρομποτικών συστημάτων πρέπει να ξεπεράσουν μια σειρά δευτερογενών προβλημάτων που ανακύπτουν πέρα από την κύρια εργασία της ανάπτυξης αλγορίθμων και λογισμικού, όπως οι μεταβολές στο υλικό του ρομπότ, η συνεχής εξέλιξη του API, προβλήματα διαμεταγλώττισης, ζητήματα απόδοσης, καθώς και η έλλειψη γενικών βιβλιοθηκών λογισμικού. Η Monas στοχεύει στην αντιμετώπιση αυτών των προβλημάτων με την παροχή ενός επιπέδου αφαίρεσης από το υλικό και ενός γενικού πλαισίου για τη σύνθεση ρομποτικών πρακτόρων. Επιδιώκοντας την αξιοποίηση καταξιωμένων μεθοδολογιών λογισμικού στο πλαίσιο της ρομποτικής, έχουμε ενσωματώσει μια μεθοδολογία AOSE (Agent Oriented Software Engineering) στην Monas. Συγκεκριμένα, η μεθοδολογία ASEME (Agent Systems Engineering Methodology) χρησιμοποιήθηκε για την ανάπτυξη του λογισμικού για μια ρομποτική ομάδα που συμμετέχει στο Standard Platform League (SPL) του διαγωνισμού RoboCup (το παγκόσμιο κύπελλο ρομποτικού ποδοσφαίρου). Η ομάδα αποτελείται από τρία ανθρωποειδή ρομπότ που παίζουν ποδόσφαιρο αυτόνομα σε πραγματικό χρόνο χρησιμοποιώντας τις on-board δυνατότητες για αντίληψη, επεξεργασία, και δράση, ενώ επικοινωνούν και συντονίζονται μεταξύ τους, προκειμένου να επιτύχουν τον κοινό στόχο τους, να κερδίσουν δηλαδή το παιχνίδι. Η εργασία μας αντιμετωπίζει κυρίως το πρόβλημα του συντονισμού των βασικών λειτουργιών του κάθε ρομπότ (αναγνώριση αντικειμένων, εντοπισμός θέσης, κινητικές δεξιότητες), προκειμένου να επιτευχθεί μια επιθυμητή ομαδική συμπεριφορά. Το πλαίσιο που προτείνουμε καταδεικνύει την προστιθέμενη αξία της χρήσης μιας μεθοδολογίας AOSE στη ρομποτική, καθώς η ASEME επέτρεψε συμπαγείς αναπαραστάσεις της κατάστασης των πρακτόρων και σύνθετες συνθήκες μετάβασης καταστάσεων, αυτοματοποίησε ένα μεγάλο μέρος της διαδικασίας παραγωγής κώδικα και μείωσε το συνολικό χρόνο ανάπτυξης των πρακτόρων.

Contents

List of Figures

List of Tables

List of Algorithms

Listings

Chapter 1

Introduction

In present days, the advancement of robotic technology has opened several possibilities for deployment in several domains, such as search and rescue, surveillance, housekeeping tasks, elderly care, etc. However, despite the improved hardware, the reduced costs, and the increased production rates of robotic systems, the development of such consumer applications is still a domain of experts. This thesis describes Monas, a flexible software architecture for the development of general-purpose robotic agents. Currently, robot software developers have to overcome a variety of problems secondary to their main task of development at the algorithmic and software level, such as hardware differences, constantly evolving robot APIs, cross-compilation issues, performance issues, and the lack of generic third-party software libraries. Monas aims at addressing these problems by providing an abstraction from the robot hardware level and a generic framework for synthesizing robotic agents.

Additionally, in the quest of utilizing principled software engineering methodologies in the context of robotics, we have integrated an Agent Oriented Software Engineering (AOSE) methodology with Monas. Specifically, the Agent Systems Engineering Methodology (ASEME) was used as it is an open-platform, compact (requires the editing of a max of four models), and met the special requirements for robotics.

For the evaluation of our approach, we developed the software for a physical robot team competing in the Standard Platform League of the RoboCup competition (the robot soccer world cup). The team is composed by three humanoid robots who play soccer autonomously in real time utilizing the on-board sensing, processing, and actuating capabilities, while communicating and coordinating with each other in order to achieve their common goal of winning the game. Our work addresses mainly the challenge of coordinating the robot's base functionalities (object recognition, localization, motion skills) in order to present a desired team behavior. The software was implemented by three different approaches and finally it demonstrates the added value of using an AOSE methodology in robotics, as ASEME allowed for compact representations of the state of the agents and complex conditional state transitions, automated a large part of the code generation process, and reduced the total development time for the agents.

1.1 Thesis Overview

Chapter 2 provides some background information on the concepts used in the thesis. This chapter is mainly intended to be a reference to the reader. Many concepts are analyzed, with most important the Statechart (Section 2.4), Agent System Engineering Methodology (Section 2.5), and the Narukom communication system (Section 2.6). Finally, the reader is introduced to RoboCup competition (Section 2.3), in which this thesis is evaluated. It is a rather introductory chapter, presenting the basic ideas, subsequently should you need any further details, please refer to the bibliography.

In Chapter 3 we discuss the problem that we aim to solve; where did we receive our inspiration, what are our expectations from a software architecture, and what were the difficulties that we faced on developing intelligent agents on robots. All these questions are going to be answered in this chapter. Also, Related Work (Section 3.4) is presented in this chapter.

Continuing to Chapter 4 the main core of this thesis is presented, which is the development of Monas architecture. The chapter is split it in three sections. The first one describes the architecture structure and utilities within it (Section 4.1). The second one, a simple way to instantiate software modules as well as with a system for agent management (Section 4.2).

Moving on Chapter 5, a discussion on the results is taking place, providing a rather empirical evaluation of our work, since it was tested, judged and compared against other team's work in this competitive domain.

Lastly, in Chapter 6, we reader will find the conclusion, and some ideas for further work.

Chapter 2

Background

In this chapter we are going to discuss concepts that already exits and are used by the implementation in this thesis. It acts both as a reference, as well as an introduction to the reader to familiarize with the various fields that this thesis combines.

2.1 Robotic Autonomous Agent

Researchers involved in the field of artificial intelligence often define the term differently, each one trying to approach the essence of the agent in their context of research. In this thesis, we specialize to autonomous agents running on robotic platforms, therefor we can define the agent as a software entity or system that inhabit some complex environment, has the ability to perceive it, and acts autonomously in it. By doing so, the agent, realize a predefined set of goals or tasks. The agent is viewed as an entity that perceiving its environment through sensors and acting upon through the use of actuators. Can be described mathematically (Equation 2.1), as a function f which maps every possible percept sequence P^* to a possible action A the agent can perform. We denote agents to be autonomous, at least to some degree,so that the agent executes independently, without any external interfering, making its decisions by himself.

$$f : P^* \to A \tag{2.1}$$

The agent can be executed in a variety of environment with different characteristics. Although the amount of possible different environments is vast, can be classified according some properties to assist researchers in the agent design process. The environment classification is done as follows:

Fully Observable or **Partially Observable.**
When the sensors of an agent provide access to the complete state of the environment at each point of time the we derive the environment as fully observable. If the environment because of sensor's noise and inaccuracy or because a part of the environment's state is not covered by sensors the environment is classified as partially observable.

Deterministic or **Stochastic.**
If the next state of the environment can be totally defined from the current state and from the action that the agent executes then the environment is deterministic; elsewhere is classified as stochastic. Whereas the environment is deterministic the agent may have the ability to partially observe it and thus appear as a stochastic environment. Hence, it is generally best to characterize the environment from the agent's point of view.

Episodic or **Sequential.**
In an episodic environment the agent's experience is divided into atomic episodes. In each episode the agent receives a percept and the decides a single action. The next episode does not does not depend on action done on previous episodes. The selection of the action depends only from the episode the agent is in. On sequential environments, on the other hand, the current decision could affect all future decisions.

Static or **Dynamic.**

An environment that has the ability to be modified while the agent is deciding its next action then the environment is characterized as dynamic. The characterization take place from the agent point of view and describe how the environment is perceived by agent and is not assumed as a mandatory characteristic of the environment itself. If the environment is not dynamic, then is classified as static.

Discrete or **Continuous.**

The separation between ta discrete and a continuous environment has to do with the environment's characteristics such as time, state and action space. If the time can be discretized, as a turn based strategy game, and the actions are also discrete, like the move of a chessman, then the environment is discrete, otherwise is a continuous environment.

Single-Agent or **Multi-Agent.**

When the same environment is inhabited from multiple agents, is characterized as a multi-agent environment, and when is inhabited by only one agent a single-agent environment. Multi-agent environment can then categorized into cooperative and competitive depending on inter-agent relationship.

Generally the agent faces more difficulties when inhabits a partially observable environment rather than a fully observable, a stochastic rather than an episodic and so forth, with the latter classification in each characteristic making the environment significantly harder for the agent to operate than the former.

2.2 Robot

Robot are electro-mechanical systems, capable for sensing its environment and, with the use of actuators, modifying it in a way to achieve a goal or perform a task. Robots are also often defined as physical agents as its usual environment is the real world and in rare cases a simulation world. The word ”robot” originate from the by Czech writer Karel Čapek in his play R.U.R. (Rossum's Universal Robots), published in 1920 [ČN04]. Robot nowadays are operated in a variety of different applications and environments from manual labor, industry, surgery and universe exploration to even intellectual tasks such as painting. Robots, depending on the application, have various forms and shapes. Sometimes mimic the nature so their form is inspired from humans, dogs, hexapods or larvas but mostly their shape is not found in nature and is designed for maximizing the robots efficiency such as the robotic arms and wheeled robots.

2.2.1 Aldebaran's Nao Humanoid Robot

Nao is a humanoid robot developed by the French company Aldebaran Robotics. It is a programmable, medium-sized robot that reach the market on the first quarter of 2008, after a tough research and development phase lasted over three years []. The first version of the robot, was deployed for academic use while the robot was also selected for the robocup competition. The platform provides a great amount of precise sensors and actuators as well as a programming framework, NaoQi named after the traditional Chinese culture, qi, which enables fast and easy access to the robot's hardware.

Nao robot is a 58 cm (23") tall biped that weights about 4.3 kgs. The Robocup edition has 21 degrees of freedom, six on each leg, four on each hand, tow for the head. The hip-yaw joint can not be controlled independently from each leg but instead is coupled so that both legs share the same hip-yaw angle decreasing the degrees of freedom to the total of 21. Nao's on-board computer is an AMD Geode system-on-a-chip microprocessor, clocked at 500MHz which supports the x86 instruction set. It occupies 256MB of Random Access Memory (RAM) and a 2GB usb flash memory for storage purposes. The power on-board is provided by a 6-cell lithium-ion (Li-ion) battery giving the robot about 90 minutes of autonomy. The robot is presented on Figure 2.1.

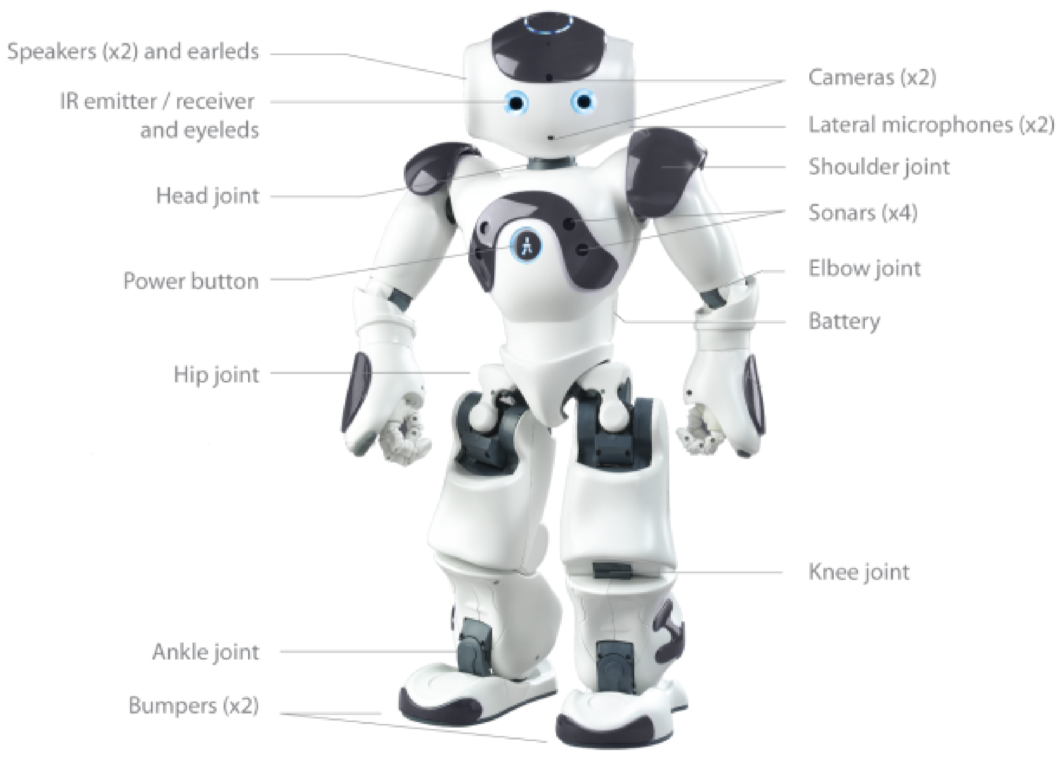


Figure 2.1: Aldebaran's Nao Robot (Robocup Edition)

The Nao's operating system Linux distribution for usage on embedded systems and is customized by Aldebaran for use with the robot. As an interface between the robotic hardware Aldebaran provides the NaoQi framework capable for controlling the robot's actuators and gather the sensors information. Except from the execution of simple control commands, NaoQi provides an interface to more complex such as Cartesian space body-part movement as well as an omni-directional walking engine. NaoQi additionally provides a Software Development Kit (SDK) that enable developing of custom applications. The SDK provide two building methodologies: one that integrates directly on NaoQi, enabling fast access on the hardware, and a broker based which is failure tolerant (as an application crash will not lead to a NaoQi crash) and can be used over the network enabling power-consuming applications to run remotely. The architecture is illustrated in Figure 2.2. Additionally, NaoQi except from the sequential (blocking) calls, provide a mechanism for parallel execution, in which the call is assign to an other thread. Event driven programming, implementing through callbacks, has a limited support. Finally, NaoQi calculates and checks if the requested commands can be safely executed and aborts the commands that can cause hardware damage.

Nao robot has a variety of sensors and actuators making it pretty competitive among humanoid robots at this price list. It occupies twenty one variable-force servo motors for executing complex movements as well as encoders in every joint to provide the accurate angle of the joint. Nao operates two CMOS digital cameras capable for video capture at a maximum resolution of 640x480 pixels at 30 frames per second (fps). Stereoscopic vision is not available because of hardware limitation which pose that only one camera can be enabled at the moment and because the cameras are mounted on the forehead and on the chin do not have an overlap area. Also two speakers and two microphones can be used for sound communication mounted at the ears of the robot. The speakers, in combination with the NaoQi's text-to-speach (T2S) capabilities, can be used for more natural communication between humans and the robot, while the two microphones provide basic functionalities for sound localization. Additionally four ultrasound sensors are placed on the robot's chest for obstacle detection.

Further more Nao provide internal inertial sensors, consisting of a 3-axis accelerometer and a 2-axis gyrometer, located at the robot's torso. Each foot occupies four force sensitive resistors which provide information on the pressure distribution over the foot area and thus enabling the calculation of the center-of-gravity (CoG). Bumpers at the front of each feet also provide detection of collisions, sometime undetectable from the ultrasound sensors. A button at the torso provide a convenient interface for starting-up and shutting-down the robot. Finally wired - ethernet and serial - as well as wireless (WiFi) adapters provide the necessary connectivity to the outer world.

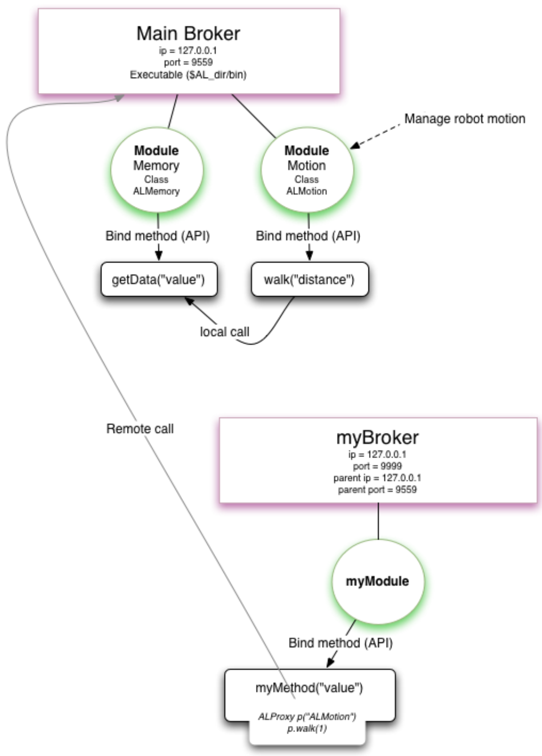


Figure 2.2: NaoQi’s middleware architecture

2.3 RoboCup Competition

RoboCup competition is an international research and education initiative. It attempts to promote Artificial Intelligence and autonomous robotics by providing a standard problem where wide range of technologies can be integrated and examined. RoboCup was founded in 1993 by Hiroaki Kitano with a bold vision: conduct a soccer game between a team of fully autonomous humanoid robots and the human world soccer champions by the year of 2050, complying with the official rule of the FIFA [KAK+97]. The name RoboCup is a contraction of the competition’s full name, ”Robot Soccer World Cup”, but despite its original name, there are many other stages of the competition such as ”Search and Rescue” and ”Robot Dancing” have been added. The RoboCup competition, while has a short history, has grown to a well-established annual event bringing together the best robotics researchers from all over the world.

Figure 2.3: RoboCup Federation's Logo and major domains

RoboCup's competitions bridges theory and practice as it poses real-world challenges that the participants must overcome. Participating requires various technologies to be incorporated that not only analyze the problem theoretically but also require a wide range of system technicalities as well. Major problem in robotics such as perception, cognition, action, and multi-agent collaboration must be efficiently solved under real-time and resource constraints. RoboCup's competitive nature exhaustively tests the proposed approaches under the same controlled conditions and promotes the best of them so to advance the state-of-the-art in the area.

The contest currently holds four major competition domains, each with a number of leagues and subleagues: RoboCup Soccer League, RoboRescue, RoboCup Junior and RoboCup@Home. RoboCup's main field of action is soccer but despite its ostensibly simplicity, is considered as one of the most difficult environments in Artificial Intelligence. As is an agent environment, it can be classified as described in 2.1, hence it has the following attributes:

Partially Observable. While playing the agent has not full knowledge of the environment due to sensor noise and error, camera restrictions and lack of computational power so it is characterized as a partially observable environment.

Sequential. A decision to move towards the ball will influence all the agents later actions making the environment sequential.

Stochastic. A kick's success or not depends from a vast amount of parameters that, although can be calculated analytically in theory, can't be in practice so the environment is considered as stochastic.

Dynamic. The environment is changing constantly, even during the time that the agent is figuring out its next move, composing a dynamic environment.

Continuous. The player is playing in a real soccer field and its position is determined by real coordinates.

Competitive Multi-Agent. The state space of the agent acts in is continuous and many more agents, both friendly and rival, act in parallel forming a multi-agent competitive and cooperative mixed environment.

To summarize, the trivial soccer environment for the human beings is for the robotic agent very difficult to percept and even more difficult to decide the next action.

2.3.1 RoboCup: Standard Platform League

Standard Platform League (SPL) belongs to the soccer domain of RoboCup competition and is one of the most popular leagues. Each participant, uses the same robotic platform to reduce the task of winning the game to the developing of efficient and sophisticated software implementations. This league was formerly known as the Four-Legged League where the common platform was Sony's Aibo quadruped robotic dog. The current, common platform is the humanoid Nao robot 2.2.1, provided by Aldebaran Robotics (Figure 2.4). The league features three vs three games, increased to four vs four for 2011, in a 4×6 meters soccer field marked with thick white lines on a green carpet. The two colored goals (sky-blue and yellow) also serve as landmarks which aid the localizing process of the robots on the field. The game consists of two 10-minute halves and teams switch colors and sides at halftime. A complex set of rules ensures the smooth flow during the game and is implemented by human presence. For example, a player is punished with a 30-seconds removal from the

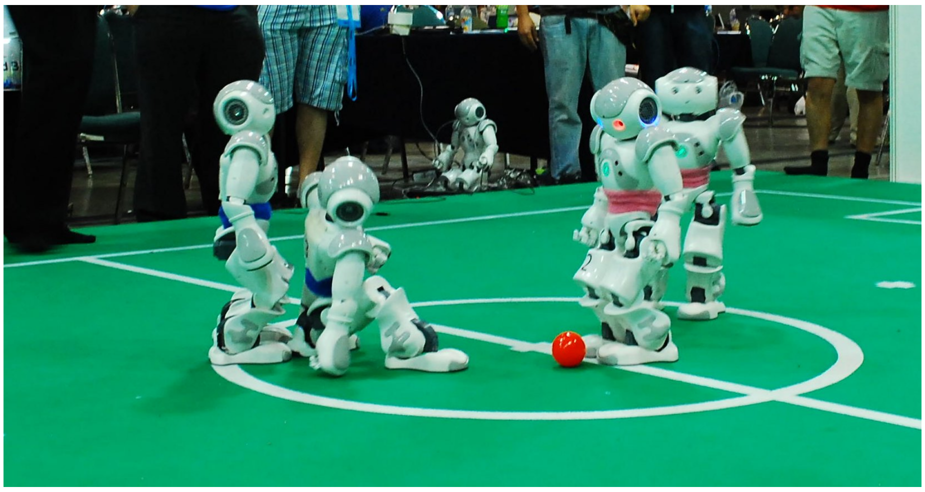


Figure 2.4: Standard Platform League game in Robocup 2010

field if he performs an illegal action, such as pushing an opponent for more than three seconds, grabbing the ball between his legs for more than three seconds, or entering his own goal area as a defender.

The main characteristic of the Standard Platform League is that no hardware changes are allowed; all teams use the exact same robotic platform and differ only in terms of their software. This convention results to the league's enrichment of a unique set of features: autonomous player operation, vision-based perception, legged locomotion and action. Given that the underlying robotic hardware is common for all competing teams, research effort has been focused on the development of more efficient algorithms and techniques for visual perception, active localization, omni-directional motion, skill learning, and coordination strategies. During the course of the years, one could easily notice a clear progress in all research directions.

2.3.2 Team Kouretes

Team Kouretes was founded in February 2006 and became active in the Four-Legged league. The team had its first exposure to RoboCup at the RoboCup

Figure 2.5: Kouretes team 2008 formation. From left to right in the front row are Andreas Panakos (SPL), Daisy Chroni (Simulation Team), Alexandros Paraschos (SPL), Stathis Vafias (Simulation Team), and in the back row Professor Michail G. Lagoudakis (Kouretes Team Leader) and Georgios Pierris (SPL)

2006 event in Bremen, Germany, where it participated in the Technical Challenges of the Four-Legged league. At that time, Aibo programming by the team was done exclusively in the interpreted Universal Real-Time Behavior Interface (URBI), without any use of existing code. Subsequent work led to the participation of the team in the Four-Legged league of the RoboCup German Open 2007 competition in Hannover, Germany. The software architecture of the team was developed on the basis of previously released code by GT2004 and SPQRL 2006. In Spring 2007, the team began working with the newly-released Microsoft Robotics Studio (MSRS). The team's software was developed from scratch exclusively in C# and included all the required services, as well as the motion configuration files for the simulated RoboDog robot of RoboSoft. The team's participation in the MSRS Simulation Challenge at RoboCup 2007 in Atlanta, USA led to the placement of the team at the 2nd place worldwide bringing the first trophy home.

In Spring 2008 the team switched to the new robotic platform, the Aldebaran Nao humanoid robot, working simultaneously on the real robots and on the Webots and MSRS simulators and developing new code from scratch. In the recent RoboCup 2008 competition in Suzhou, China the team participated in all divisions of the Standard Platform league (Aibo robots, Nao robots, Nao Webots simulation, Nao MSRS simulation). The team's efforts were rewarded in the best possible way: 3rd place in Nao league, 1st place in the MSRS simulation, and among the top 8 teams in the Webots simulation (Figure 2.5).

2.4 Statecharts

Finite state machines (FSM) are computational models that consist of a set of states, an initial state, an input alphabet and a transition function that maps every legal state combination to an other legal state combination, given an input symbol. Hence, FSMs, "specifies the sequence of states an object goes through during its lifetime in responses to events, together with its responses to those events" [BRJ99]. FSMs achieve better results from textural representations when describing reactive rather than transactional systems.

Statecharts are state diagrams, very useful for behavioral modelling. They differ from other forms of state diagrams, such as the classical finite state machines and its derivatives, because they address two major problems that mainly affect the number of nodes and transitions: hierarchy and orthogonality. Additionally, statecharts incorporate a very power visual representation which improves the readability and understanding by the reader.

Statechart does not have a single formalism but instead have three, that appear to be very similar. Historically, the first one is *Classical Harel's* statecharts (as being implemented in *Statemate* [HN96]), while the other two were developed almost concurrently —borrowing elements from each other —are the object-oriented version of Harel's statechart (implemented in *Rhapsody* [HK04] tool) and the *UML State Machine Diagrams*, specified in [Gro05]. In this thesis, the formalism that is followed is a modified version of `Rhapsody` statecharts (each difference is stated explicitly).

There are three types of *states* in a statechart [HK04], i.e. OR-states, AND-states, and basic states. *OR-states* have sub-states that are related to each other by "exclusive-or ", and *AND-states* have orthogonal components that are related by "and "(they are executed in parallel). *Basic states* are those at the bottom of the state hierarchy, that is those that have no sub-states. The state at the highest level (the one with no parent state) is called the root. The active states at a specific time, consist the active configuration of the statechart.

The execution flow is decided from the transitions between the states. Each *transition* from one state (source) to another (target) can be labeled by an *expression*, whose general syntax is $e[c]/a$, where e is the event that triggers the transition; c is a condition that must be true in order for the transition to be taken when e occurs; and a is an action that takes place when the transition is taken. All elements of the transition expression are optional. A transition with an empty transition expression, all three parts missing, is called a null transition. Moreover, there are *compound transitions* (CT). These transitions are sequences of transition segments, connected by special states (defined as *connectors*) between a source and a target state, or form an other point of view, transitions that can have more than one source or target states. There are two kinds of CTs: *AND-connectors* and *OR-connectors*. AND connectors are of two types, joint transitions (more than one sources) and fork transitions (more than one targets). The most commonly used OR-connector is the conditional transition. The *scope* of a transition is the lowest level OR-state that is a common ancestor of both the source and target states. When a transition occurs all states in its scope are exited and the target states are entered.

Additionally, two more categories of states exist to help the realization of specific behaviours on statecharts: *psedo-states* and *transition connectors*. In the former category we can locate *START* and *END* states, witch represent the initial transition and a sink (a state with no outgoing transitions). In the latter category we can find out states that are used on compound transitions such as the *junction, condition, fork and join* connectors.

As being defined for FSMs, statechart are changing configurations given an event. Then, none, one or more transitions (or compound transitions) are activated and change the active configuration of the statechart, leaving it in a legal —statecharts can never "stop"their execution in the middle of a *transition segment*, a *psedo-state*, a *connector* or by activating a *composite state* and not it's substate —and stable (no more null-transitions can be executed) configuration.

Problems arise when more than one transitions can be executed at a specific execution step, but each one leads to a different active configuration. The point is crucial as if the two or more transitions are in different scopes, the one with the lower scope has priority, but if the transitions are in the same scope, then we arbitrary select one (the selection depends of the implementation).

Multiple concurrently active statecharts are considered to be orthogonal components at the highest level of a single statechart. If one of the statecharts becomes non-active (e.g. when the activity it controls is stopped) the other charts continue to be active and that statechart enters an idle state until it is restarted.

2.5 THE AGENT SYSTEMS ENGINEERING METHODOLOGY

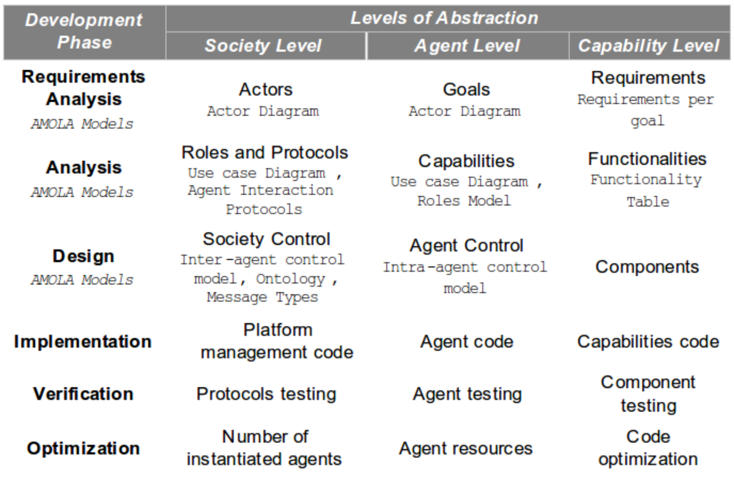
The Agent Systems Engineering MEthodology (ASEME) [SM10] is an Agent Oriented Software Engineering (AOSE) methodology for developing multi-agent systems. It uses the Agent MOdeling LAnguage (AMOLA) [SM08], which provides the syntax and semantics for creating models of multi-agent systems covering the analysis and design phases of a software development process. It supports a modular agent design approach and introduces the concepts of intra- and inter-agent control. The former defines the agent's behavior by coordinating the different modules that implement his capabilities, while the latter defines the protocols that govern the coordination of the society of the agents.

ASEME applies a model driven engineering approach to multi-agent systems development, so that the models of a previous development phase can be transformed to models of the next phase. Thus, different models are created for each development phase and the transition from one phase to another is assisted by automatic model transformation, including model to model (M2M), text to model (T2M), and model to text (M2T) transformations leading from requirements to computer programs. The ASEME Platform Independent Model (PIM), which is the output of the design phase, is a statechart that can be instantiated in a number of platforms using existing Computer Aided System Engineering (CASE) tools.

2.5.1 ASEME Process

ASEME [SM10] specifies the entire software development process for the development of agents. A *Software Process* is defined as a series of *Phases* that produce *Work Products*. In each phase simple or complex *activities* take place. Simple activities are defined as *Tasks*. Activities are achieved by *Human Roles*. Work products can be either graphical or textual models. Graphical models can be *Structural* (focusing in showing the static aspects of the system—such as class diagrams) or *Behavioral* (focusing on describing the dynamic aspects of the system—what happens as time passes). Textual models can be completely free text or follow some specifications or a grammar.

Three levels of abstraction are defined for each phase. The first is the *societal level*, in which the whole multi-agent system functionality is modeled. Then, the *agent level* zooms in each member of the society, i.e. the individual agent. Finally, the details that compose each of the agent's parts are defined in the *capability level*. The concept of *capability* is defined as the ability of an agent to achieve specific tasks that require the use of one or more *functionalities*. The latter refers to the technical solution(s) to a given class of tasks. Moreover,



Development Phase	Levels of Abstraction		
	Society Level	Agent Level	Capability Level
Requirements Analysis *AMOLA Models*	Actors Actor Diagram	Goals Actor Diagram	Requirements Requirements per goal
Analysis *AMOLA Models*	Roles and Protocols Use case Diagram , Agent Interaction Protocols	Capabilities Use case Diagram , Roles Model	Functionalities Functionality Table
Design *AMOLA Models*	Society Control Inter-agent control model, Ontology , Message Types	Agent Control Intra-agent control model	Components
Implementation	Platform management code	Agent code	Capabilities code
Verification	Protocols testing	Agent testing	Component testing
Optimization	Number of instantiated agents	Agent resources	Code optimization

Figure 2.6: ASEME phases and their AMOLA products.

capabilities are decomposed to simple *activities*, each of which corresponds to exactly one functionality. Thus, an activity corresponds to the instantiation of a specific technique for dealing with a particular task. ASEME is mainly concerned with the first two abstraction levels assuming that development in the capability level can be achieved using classical (or even technology-specific) software engineering techniques.

In Figure 2.6, the ASEME phases, the different levels of abstraction, and the models related to each one of them are presented. Some of the products (like the ontology product of the design phase) are not AMOLA models. In these cases, classical software engineering models can be used. In the same figure the reader can see the human roles that are expected to work at each phase.

2.5.2 AMOLA

AMOLA [SM08] describes both an agent and a multi-agent system. Before presenting the language itself, some key concepts must be identified. Thus, the concept of *functionality* is defined to represent the sensing, thinking, and acting characteristics of an agent. Then, the concept of *capability* is defined as the ability to achieve specific goals (for example, the goal of deciding which restaurant to dine in tonight) that requires the use of one or more functionalities. The capabilities are the modules that are integrated using the *intra-agent control* concept to define an agent. Individual agents are integrated to form a multi-agent system using the *inter-agent control* concept.

The AMOLA models are related to the requirements analysis, analysis, and design phases of the software development process. AMOLA aims to model the agent community (by defining the protocols that govern agent interactions), as well as the individual agents (by defining the agent capabilities and the functionalities for achieving them). The agent's functionalities are defined using classical software engineering techniques.

In the requirements analysis phase, AMOLA defines the *System Actors and Goals* (SAG) model, containing the system's actors and their goals. In the analysis phase, AMOLA defines the *System Use Cases* (SUC) model, where the different activities that realize the agent capabilities are defined in a top-down decomposition process, the *Agent Interaction Protocol* (AIP) model and the *System Roles Model* (SRM), through which the previously defined activities are connected to define the dynamic behavior of the roles, and the *Functionality Table* (FT), which is mainly used by project managers to select the various technologies that will be used for the project implementation. In the design phase, AMOLA defines the *intEr-Agent Control* (EAC) model and the *Intra-Agent Control* (IAC) model, which are based on the formalism of *statecharts* [HK04] and define the functional and behavioral aspects of the multi-agent system. EAC defines interaction protocols by specifying the necessary roles and the interactions among them. The implementation of EAC is realized at the agent level via the IAC, which specifies the capabilities and their appropriate interaction. Finally, each capability is defined with respect to the required functionalities, the technology used, the parametrization, and the implemented data structures and algorithms. As a side note, IAC corresponds to the Platform Independent Model (PIM) level of the Model Drive Architecture (MDA) [KWB03].

AMOLA Metamodels

Use case model (SUC).

In the analysis phase, the analyst needs to start capturing the functionality behind the system under development. In order to do that he needs to start thinking not in terms of goal but in terms of what will the system need to do and who are the involved actors in each activity. The use case diagram helps to visualize the system including its interaction with external entities, be they humans or other systems. It is well- known by software engineers as it is part of the Unified Modeling Language (UML). In AMOLA no new elements are needed other than those proposed by UML, however, the semantics change. The actor "enters" the system and assumes a role. Agents are modeled as roles, either within the system box (for the agents that are to be developed) or outside the system box (for existing agents in the environment). Human actors are represented as roles outside the system box (like in traditional UML use case diagrams). This approach aims to show the concept that we are modeling artificial agents interacting with other artificial agents or human agents. Finally, the different use cases must be directly related to at least one artificial agent role.

Role model (SRM).

An important concept in AOSE is the role. An agent is assumed to undertake one or many roles in his lifetime. The role is associated with activities and this is one of the main differences with traditional software engineering, the fact that the activity is no longer associated with the system, rather with the role. Moreover, after defining the capabilities of the agents and decomposing them to simple activities in the SUC model we need to define the dynamic composition of these activities by each role so that he achieves his goals. Thus, we defined the SRM model based on the Gaia Role model [WJK00]. Gaia defines the liveness formula operators that allow the composition of formulas depicting the role's dynamic behavior. However, we needed to change the role model of Gaia in order to accommodate the integration in an agent's role the incorporation of complex agent interaction protocols (within which an agent can assume more than one roles even at the same time), a weakness of the Gaia methodology. The SRM metamodel defines the concept Role that references the concepts:

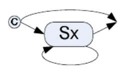
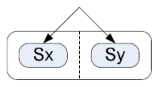
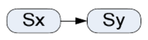
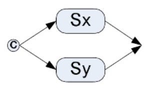
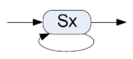
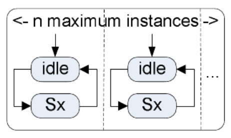
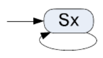
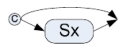
- *Activity*, that refers to a simple activity with two attributes, name (its name) and functionality (the description of what this activity does),
- *Capability*, that refers to groups of activities (to which it refers) achieving a high level goal, and,
- *Protocol*. The protocol attributes name and participant refer to the relevant items in the Agent Interactions Protocol (AIP) model. It is used for identifying the roles that participate in a protocol, their activities within the protocol and the rules for engaging.

The final formula can be described briefly as, denote two activities, A and B, $A.B$ means that activity B is executed after activity A, A^{ω} means that activity A is executed forever (ti restarts as soon as it finishes), $A|B$ means that either activity A or activity B is executed and $A||B$ means that activities A and B are executed in parallel. Additionally to the Gaia operators, a new operator is introduced: $|A^{\omega}|^{n}$ which define an activity can be concurrently instantiated and executed more than one times (n times).

IAC Metamodel.

In our work we use statecharts to model both IAC and EAC. As we said before, it corresponds to modeling the interaction between different capabilities, defining the behavior of the agent. This interaction defines the interrelation in a recursive way between capabilities and also between activities of the same capability that can imply concurrent or sequential execution. This is the basic and main difference with the way that statecharts have been used in the past. Moreover, we use statecharts in order to model agent interaction, thus using the same formalism for modeling inter and intra- agent control, which is also a novelty.

Table 2.1: Templates of extended Gaia operators for Statechart generation

Op.	***Template***	***Op.***	***Template***	***Op.***	***Template***
x^*		$x \parallel y$		$x . y$	
$x \mid y$		$x+$		$\lvert x^{\omega} \rvert^{n}$	
x^{ω}		$[x]$			

In the agent level, we define the intra-agent control by transforming the liveness model of the role to a state diagram. We achieve that, by interpreting the Gaia operators in the way described in Table 2.1. Initially, the statechart has only one state named after the left-hand side of the first liveness formula of the role model (probably named after the agent type). Then, this state acquires sub-states. The latter are constructed reading the right hand side of the liveness formula from left to right, and substituting the operator found there with the relevant template in Table 2.1. If one of the states is further refined in a next formula, then new sub-states are defined for it in a recursive way.

At the society of the agents, we define the inter-agent control (EAC) model. The EAC is a statechart that contains an initial (START) state, an AND-state named after the protocol, and a final (END) state. The AND-state contains as many OR-states as the protocol roles named after the roles. Two transitions connect the START state to the AND state and the AND state to the END state. The transition expressions are defined in EBNF format defined by the user. Transitions can be triggered by a timeout event or by the ending of the executing state activity.

Both EAC and IAC models are defined by the IAC metamodel. The IAC metamodel, shown in Figure 2.7, is defined in ecore format [BBM03] and defines a *Model* concept that has *nodes*, *transitions*, and *variables* references.

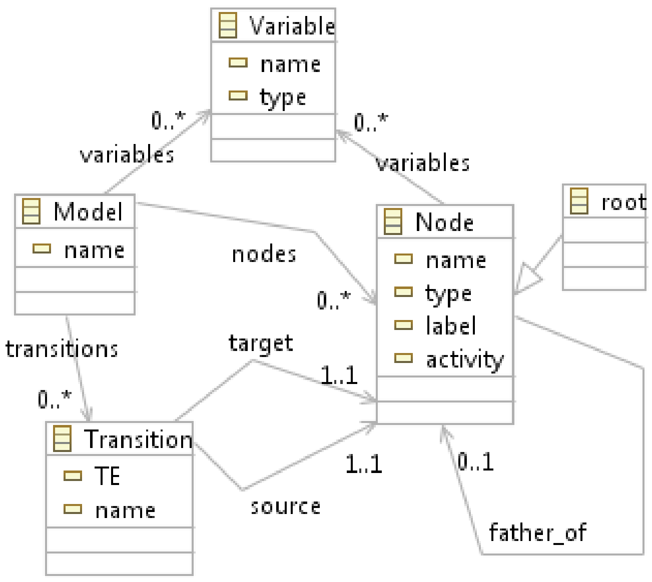


Figure 2.7: The IAC Metamodel

Note that it also has a *name* attribute. The latter is used to define the namespace of the IAC model. The namespace should follow the Java or C# modern package namespace format. The nodes contain the following attributes:

- *name,* the name of the node,
- *type,* the type of the node, corresponding to the type of state in a statechart (AND, OR, BASIC, START, END),
- *label,* the label of the node, and
- *activity,* the activity related to the node.

Nodes also refer to *variables*. The Variable concept has the attributes *name* and *type* (e.g. the variable with name “count” has type “integer”). The next concept defined in this metamodel is that of *Transition*, which has four attributes:

- *name,* usually in the form “<source node> TO <target node>”,

- *TE,* the transition expression, which contains the conditions and events that make the transition possible and through which the modeler defines the control information in the IAC. TEs can use concepts from an ontology as variables. Moreover, the receipt or transmission of an inter-agent message can be used in a TE (in the case of agent interaction protocols),
- *source,* the source node, and
- *target,* the target node.

2.6 Narukom Communication Framework

Narukom communication framework, tries to address the communication needs for inter- and intra- robot communication as well as robot-to-computer communication. It is a message-based architecture and provides a simple, efficient yet flexible way of exchanging messages between robots, without imposing restrictions on the type of the data transferred over the network. The framework is based on the publish/subscribe paradigm and provides maximal decoupling not only between nodes, but also between threads on the same node.

Narukom uses Google Protocol Buffers for the creation of its messages. Protocol Buffers are a way of encoding structured data in an efficient, flexible yet extensible format and is being used by Google for almost all of its internal Remote Procedure Call protocols and file formats. Data serialization and un-serialization which is needed for network transmissions, especially between different platforms, is carried out by Protocol Buffers.

Each transmitted message has a topic identifier which determine where the message is going to be delivered. Receivers must subscribe the topics that are interested in, in order to receive communication messages. The topic field in a message, is set by the message sender and is published across the network.

Additionally, each message comprise useful meta-data that are being transfered across the network. The meta-data contains the sender node name, the

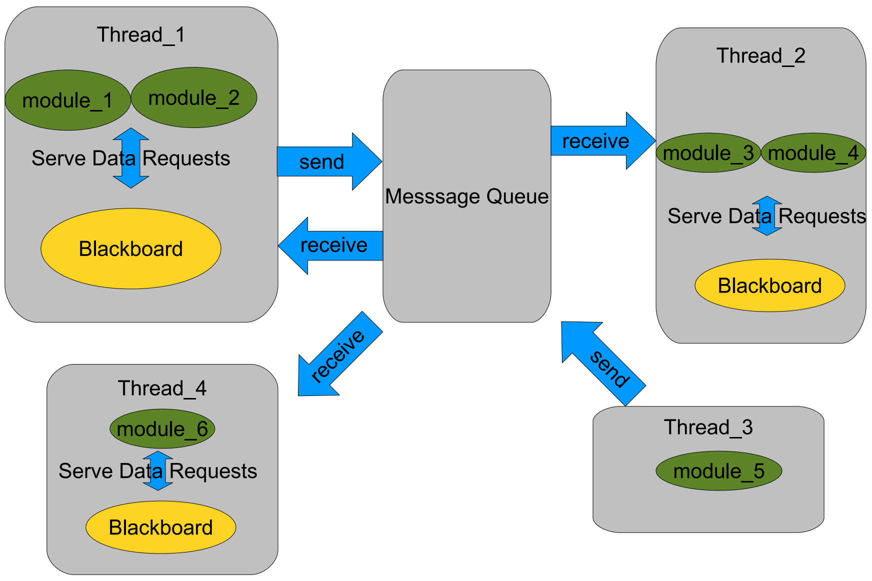


Figure 2.8: Narukom architectural model

publisher name as well as integrated temporal information address synchronization needs.

Moreover, Narukom provide a blackboard to the publish/subscribe architecture. Blackboard is a software architecture model, in which multiple individual share a common knowledge base. Individuals can read or update the contents of the blackboard and therefore cooperate to solve a problem. Is common for blackboards to organize the containing knowledge as efficient as possible to enable quick retrieval of data. Blackboard in Narukom is available only between individuals that run on the same thread of execution and provides full access, read/write, on local information and read-only access to information that arrives from third-parties.

2.7 Implementation Languages

This section presents a brief description of the programming languages used for implementing, configuring and auto-generating code in Monas software architecture. This is not an exhaustive apposition, how could be, of all the language semantics and constructs which appear in Monas implementation but it provides an introduction to the key constructs most often encountered in day-to-day use.

2.7.1 C++

C++, pronounced ”cee plus plus”, is a general-purpose programming language, developed by Bjarne Stroustrup in 1979 at Bell Labs as an enhancement to the C language. C++ was originally named C with Classes and was renamed to C++ in 1983. C++ is statically typed, which means that type checking performs at compile time rather than run-time, free-form, as the positioning of the characters in the listing text are not significant to the language and multi-paradigm, as both purely functional or purely object-oriented programs can be implemented. It also combines both high-level and low-level language features as the developer can write from classes to assembly instructions. C++ is a compile language, but it produce efficient and portable executables and avoids features that are platform specific or not general purpose.

C++'s standard library provide a collection of classes and functions which assist the developer to manage input-output, collections of data as lists, sets, maps etc. and manipulate data with wildly spread algorithms.

Additionally C++ support templates, which enable generic programming. Templates are evaluated, and -in a way- executed, by the compiler, enabling developers to run compile-time computations. Templates are a powerful mechanism that can be used for generic programming. Indeed are so powerful that a new term is used to describe their functionality, template metaprogramming. Templates are proof to be Turing-complete, meaning that any computation which is expressible by a computer program can be computed by a template metaprogram.

As C++ is one of the most popular programming languages ever created, is widely used in many environments and systems including embedded and high-performance software making it ideal for robotics. C++ has influence many popular programming languages as C# and Java.

2.7.2 Extensible Markup Language (XML)

Extensible Markup Language (XML) is a language for for encoding documents in both human and machine readable form. The language semantics are defined in the XML 1.0 Specification. XML's goals emphasize in simplicity, generality, and usability. XML is a very structured and convenient format. It is a textual data format capable for representation of arbitrary data structures. As being a markup language, special characters are used to distinguish the markup from the content. These characters are ”<” and ”>”, and every string that belongs to the markup must begin and end with them accordingly. Character strings that does not belong in markup shape the content of the file.

Tag, as being a markup construct, starts with a ”<” and ends with ”>”. Tag come in three different types: start tag, end tag and empty-element tag. The first two types always must form a pair, in order to maintain a valid XML schema, whereas the third one does not. The special character ”/” is used to disambiguate the different tag types, as its absence forms a start tag (<aTag>), the use right after the ”<” form an end-tag (</aTag>) and the use before the ”>” character an empty-element tag (<emptyElmTag/>).

Elements are components of a document that belong in a tag. They are written between a start-tag and an end-tag and can contain markup as well as content. The markup constructs inside an element tag are modelled as children of the element, forming that way a tree structure. Obviously empty element tags can not contain any elements.

Attributes are constructs consisting of a name/value pair that exists within a start-tag or empty-element tag. Multiple attributes can exists within the tag. The difference between data elements and attributes is semantically: the element is used to store the data whereas the attributes is used to store the meta-data or information that characterize the tag that the attribute belongs.

Comments may appear anywhere in an XML document and are being identified by the ”<!–”, as the start tag for the comment and the ”–>” which signals its end. Markup can also be enclosed in a comment section without the need of escaping. Nested comments are not valid in XML syntax.

XML files rarely are read directly by the application that is using them but instead an application programming interfaces (API) is used to process the data. The processor analyzes the markup and passes structured information to an application. Many application programming interfaces have been developed in order to process XML data. The processor is often referred as an XML parser.

A document in addition to being well-formed, needs to be valid. The document, in order to check its validity, must contain a reference to a Document Type Definition (DTD) that describes the schema of the file. The schema determines the set of elements that may be appear in the document, which attributes may be applied in each case and the allowable parent/child relationships. XML processors are responsible to classify the document as valid or not and report error if occurred.

2.7.3 XPand Language

XPand language was proposed by Open Architecture Ware (oAW) and is used for Model-to-Text (M2T) transformations. The language is offered as part of the

Eclipse Modeling Framework (EMP) [BBM03], a project which provides a unified set of modelling frameworks, tooling and standard implementations. The language allows the developer to define a set of templates that transform objects that exist in an instance of a model into text. Major advantages of XPand are the fact that it is source model independent, which is usually source code but it can be whatever text the user desires, and its vocabulary is limited, allowing for a quick learning curve.

The language requires as input a model instance, the model and the transformation templates. XPand first validates the instance through the provided model and then, as the name suggests, expands the objects found in the instance with the input templates. It allows the user to define, except form the expansion templates, functions implemented in Java language using the Xtext functionality.

XPand is a markup language and uses the ”$\ll$” and ”$\gg$” to mark the start and the end of the markup context. Enables code expansion using the model structure (i.e. expanding all child elements of a specific type inside a node) and supports if-then-else stracture. Functions call be called inside markup.

The advantages of Xpand are the fact that it is source model independent, its vocabulary is limited allowing for a quick learning curve while the integration with Xtend allows for handling complex requirements. Then, EMP allows for defining workflows that can help a modeler to achieve multiple parsings of the model with different goals.

2.8 TinyXML Parser

TinyXML is a simple XML parser, implemented in C++ that is easy to integrate into other programs. It solves the text I/O problem, as it enables to users to read and write XML files. TinyXML is an open source project and was mainly developed by Thomason Lee. Aims to be a simple, basic parser that is easy-to-use.

The parser builds from the XML file a Document Object Model (DOM) that can be read and modified, and also can be saved back in the XML form. It also supports construction of an XML document from scratch by adding the appropriate C++ objects to the DOM tree. TinyXML is designed to be easy and fast to learn and that is why it keeps the application programming interface as simple as possible.

TinyXML fully supports UTF-8 encoding and does not pose any special requirements on the compliant system, as does not rely on exceptions, RTTI or even the STL as it can be compiled without. The parser is relatively small in size, making it candidate for low-power systems.

As a disadvantage, the parser does not support the use of DTDs and thus XML files can not be verified according to their schema.

2.9 CMake Building System

CMake is an open-source, cross-platform building system. CMake differs from transitional building systems in the way that does not control the whole building process but instead it generates native makefiles and workspaces that can be used in the compiler environment of your choice. Performs the crucial task of managing the build process in a compiler independent fashion. Controls the software compilation process using multiple, simple, platform and compiler independent configuration files which collectively form the standard build files. A useful feature is that it allows out-of-source building, so that the object files and the source are not under the same folder, and support concurrent building for different platforms. CMake except from the default builders for common languages, supports custom builders improving the flexibility of the architecture. Additionally possesses the capability of compiling source codes, creating libraries, building executable files in arbitrary combinations, by the specified commands regardless the platform and the operating system. Supports both static as well as dynamic library builds, as also a ”module” built which supports `dlopen` functionality.

At first, CMake search for `CMakeLists.txt` file at the specified directory and then follows the directory stracture and includes configuration files into sub-folders through the `add_subdirectory` directive. Complex directory hierarchies are supported as well as applications dependent on several libraries. It can also handles projects consisting of multiple libraries, wherein each library might be spread over several directories. Options and parameters can be passed into the system so to modify the building process depending on the user needs. A graphical editor exists to make the handling of the options and arguments easier.

2.10 Software Design Patterns

Software design patters, as formally declared, are solution that boost source code reusability and architectural design to commonly occurring problems in software engineering. Design patterns are templates, or descriptions, which provide a method for solving a problem on a range of different situations. Can be transformed to source code with ease, but, generally, are not a finished design. Patterns differ from algorithms as former solve issues that arise on the software design whereas the latter solve computational problems.

2.10.1 Singleton Pattern

Singleton pattern is a design pattern that is used when restricting the instantiation of a class to exactly one object is applied. The pattern does not create necessary the object at initialization but it is possible the object to be created on first use. It is generally useful when exactly one object is needed to coordinate actions across the system, as in case of an object factory. The concept of the singleton is sometimes generalized in order to specify an upper limit on the number of objects created.

An implementation of a singleton pattern must asserts the unity of the instance and provide a global point to access it. A singleton is an improved global variable. Unlike global variables, the singleton pattern not only provides that global access point but also restricts the instantiation of more objects of the same type. To satisfy the object uniqueness that the pattern demands, the constructor and copy-constructor are implemented as a private or protected members. The copy-constructor must be explicitly declared in order to avoid a public default implementation declared by the compiler. Additionally the mechanism to access the singleton class globally can be achieved by implementing a public static method that returns the instance of the class. The static keyword provides the global access point to the rest of the source code. The method is also responsible for managing the single instance lifetime: although a singleton can be implemented simply as a static instance so that the compiler manages the construction and destruction of the instance, or it can also be lazily constructed, so that no memory or resources are required until needed but leaving the method to destroy the instance correctly at the termination of the application.

2.10.2 Factory and Generic Factory Pattern

The factory pattern is software engineering pattern that is involved the process of object creation. Object creation often is not a simple process and may not be appropriate to include it within the final object itself as may lead to significant duplication of code or require information not accessible to the object. Factories, as the name suggest, act as producers of a specific type of objects. Can take advantage of class hierarchies to produce objects of different types but of the same base class, as a factory could produce different variations of a product. Additionally, the factory can manage special cases that concerns the object but should not be handled by it on the build-up and tear-down of the object. In other words, a Factory is the location in the code at which objects are constructed.

The intent in employing the pattern is to insulate the creation of objects from their usage. This allows for new derived types to be introduced with no change to the code that uses the base class.

Factories introduce an abstraction layer and help in the aliquot of the whole design and can aid statically typed, compile language, as C++, to emulate the concept of virtual constructors, whereas the information of the type of the object that will be created is known at runtime. The client does not know which concrete objects it gets from each of these factories since it uses only the generic interfaces of their products. This pattern separates the details of implementation of a set of objects from their general usage. The generic factory pattern provides a unified theme that encapsulate a group of individuals factories. Generic factories are parametrizable on the product that is going to be created and support different number of arguments on the products constructor.

2.10.3 Policy Based Design

Policy based design is a pattern originally developed for C++ language based on a language idiom and was introduced by Andrei Alexandrescu in [Ale01]. The pattern uses C++ template metaprogram to enable compile-time selection of an algorithm in a class. Although the technique is generic, and could be applied to other languages as well, it's strong bind with the particular feature set of that language is a disincentive factor.

The pattern consists from a class template, typically named as the host class, that is taking several type parameters as input, parameters that define the class behaviour. These types are called policy classes and are defined by the developer to implement a particular implicit interface. Policy classes encapsulate an orthogonal aspect of the behaviour of the host class, and the host class, by deriving them acquires their functionality. Providing a host class with a number of different policy's implementations enable an exponential number of different combinations, each defining a different class behaviour. The advantage

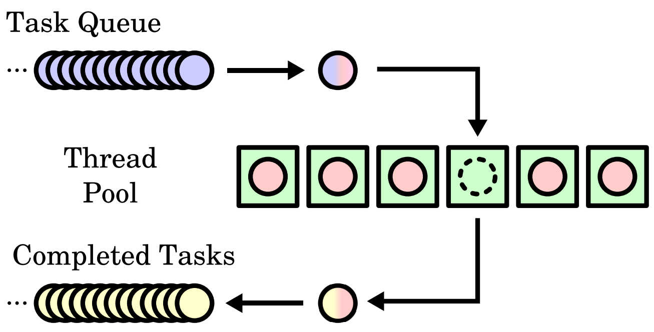


Figure 2.9: Architecture of the Thread Pool pattern

of this, decomposition, approach is that the source code has increased modularity and expandability as well as better class hierarchy design as it highlights the orthogonal components that the class consists. Additionally, the class ”assembly” is done at compilation time, thus the additional flexibility is introduced without any run-time overhead.

2.10.4 Thread Pool Pattern

The thread pool pattern serves the problem where a number of tasks are assigned to a number of threads which are perform them. It is common that incoming tasks are organized in a queue, with the queue's insertion policy varying from FiFo, LiFo and priority queue but other insertion schemes are not excluded. At the completion of the task, the thread request the next task from the queue, if exists, or sleeps otherwise. A graphical representation of the thread pool can be seen on Figure 2.9. Creating and destroying a thread and its associated resources is an expensive process in terms of time, which is the reason that thread pools generally performs better from dedicated threads, when the tasks are relatively small, as the pool's threads are initialized once and are not destroyed after the completion of the task but instead they request an other task. Hence, the creation and destruction overhead is negated.

The number of threads the thread pool occupies is parameterized and can be tuned to provide the best performance. Some implementations provide a variable number of threads, calculated dynamically, on the number of waiting tasks. The total number of threads, may have a great impact on the system's performance as too many waste valuable resources and time for their creation, whereas too few may cause significant latency from the task's arrival to the start of the tasks execution. Caution should be given not to create too many threads as the throughput of the system will significantly decrease.

Thread pools can boost an applications flexibility and scalability as the applications can be run on single CPU cores systems to multi-core almost without any modification, automatically improving the performance on the underlaying system. Additionally, developers tend to write cleaner and easier maintainable source code as it separates the task execution code from the thread management code. Finally, thread pools are straightforward on their implementation, with the only point that should be taken in notice, being the thread-safety of the queue.

Chapter 3

Problem Statement

3.1 Robotic Software Architectures

Since early ages of robotic development, designers wrote software to control their robotic creations. The software needed to match the specific requirements that robotic applications pose and the lack of available tools for this filed increased the difficulty of developing robotic software. Developers used to program the robot, to achieve a specific task, in a very verbose way, by writing source code from top to bottom and often had the source code in, a considerable number of small, pieces that neither could be characterized as modules nor could communicate. As a result, the robot could not be control in an efficient way, limiting its capabilities not to due hardware constrains but due to improper software implementations. Organizing the source code and standardize the development procedure are two major problems that architectures must address.

First of all, software architectures must solve the problem of scattered source code by introducing a modular approach on the design. They should provide the appropriate mechanics for modular programming, enabling developers to divide the problem efficiently into small fragments that can execute and co-operate into the architecture. Consequently, the source code acquires a structure and get organized decreasing maintenance and improving development times.

Additionally, an other major problem is the communication between code fragments. Communication can vary from simple cases, in which a message must be delivered in the same executable object to extensive inter-object communication in different network nodes, on different operating systems and system architectures. Thus a robotic software architectures should cope with the problem and provide an efficient and transparent way for communicating. Consequently, it will allow developers to implement distributed algorithms, which has two mainly effects. Firstly, it will enable high-level robot coordination and secondly, it will allow developers to use dedicated, powerful systems to support the on-board computer and increase the computational power.

Building source code for the target platform, usually the robot, can be a difficult and time-consuming process that distract developers from their main task. The process, in most of the cases, engage a cross-platform compiler that either is not provided by the robot's manufacture or, although that is provided, requires additional configuration. Architectures must address the case and provide an easy-to-use interface that covers the whole building process and free developers from the task.

Furthermore, developers working with physical robots, often attribute decrease on developing times on the lack of system and external libraries for the target platform, while building them on their own is very problematic. Robotics architectures should address the problem and provide building capabilities for external source code, that does not require developers to know how to configure the compilation and linking process but instead build the external source code as it would normally do within the architecture.

Finally, a robotic software architecture should be support execution on personal computers, as it will enhance both the debugging and testing procedure, as well as it will speedup the developing process.

3.2 Agent Developing

An other major problem is the robotic agent development. Developers, usually, do not follow a methodology for the creation of the agent and the main reason is the is not easy to incorporate them with the underlaying framework, or if they do, does not control the robot efficiently. The architecture should be designed to enable integration with existing methodologies or even provide one. The effects of a standardized procedure for agent creation, as we present on Chapter 5 are profound.

3.3 Robot Independence

In software engineering, and engineering in general, there is a saying that each problem must be solved only once. In our case, can be interposed as the following: developers should be able to implement a generic algorithm and run it irrespectively of the underlaying robotic system. To achieve that, a good design of a robotic software architecture should be transparent to robot changing, which means that a replace of the robotic platform should reflect form little to no changes at the source code. The developer then, is able to implement high level generic algorithms and compare their performance directly on different platforms, with ease. To achieve that, the architecture should not only be capable of compiling on different platforms, but also is important to have the robotic sensors and actuators modelled as well, in order to support transparency on changes in the robot's API.

3.4 Related Work

Murray [Mur03] proposes the use of extended statecharts for defining the behavior of RoboCup players. He designs the agent based on what he calls “modes”. These, however, need to be executed in parallel with sensors and actuators an issue not explained adequately. His work supports semi-automatic code generation for Robolog, a robot programming language based on Prolog. The proposed methodology ha been tested only in RoboCup simulation leagues, but not on real robots.

Gascueña and Fernández-Caballero [GFC09] used the Prometheus methodology [WP04] to design a robot. Their approach models each sensor and actuator of the robot as an agent, whereas in our work these are considered as functionalities coordinated using the intra-agent control concept. In Prometheus, the authors use the terms of functionality and capability. However, they are not used as independent terms. In fact, functionalities and capabilities refer to the same concept as it evolves through the development phases (i.e. the abilities that the system needs to have in order to meet its design objectives). The support for implementation, testing, and debugging of Prometheus models is limited. Another limiting issue of the methodology is the fact that the protocols definition using AIP diagrams is not used later somehow formally at the agent level. This means that the developer has to undertake the mental task of transforming the AIP diagrams to processes. The authors propose that process diagrams are to be developed by looking at the protocols involving the agent in question, as well as the scenarios developed and the goals of the agent. This is an issue that almost all the AOSE methodologies suffer from: the lack of a systematic way to integrate interaction protocol specifications to the agent capabilities.

Chapter 4

Our Approach

4.1 Software Architecture

4.1.1 Introduction

In our approach, we visualize the robot as a collection of agents. Agents are running concurrently, having their own goals. They can use any information available not only on the robot itself, but also information available on the robot's environment, for example other connected robots and computers. Thus, Monas framework aims to manage the robot's agents, allocate the resources appropriately and provide a developing process for composing agents. Additionally, Monas provide the necessary platform independence, in both robots and computers, improving source code reusability among with software tools that ease the developing of agents components. Composing the above, makes clear the architecture's name: Monas. According to the ancient Greek philosophers, the Pythagoreans, Monas represents the first being, the indivisible, but also the totality of all beings. Its symbol, a circle with a point at its center, is also been used in astronomy, symbolizing the sun, as well as by alchemists to represent gold. Hence, Monas, or μονάς in Greek, as software architecture for robotic

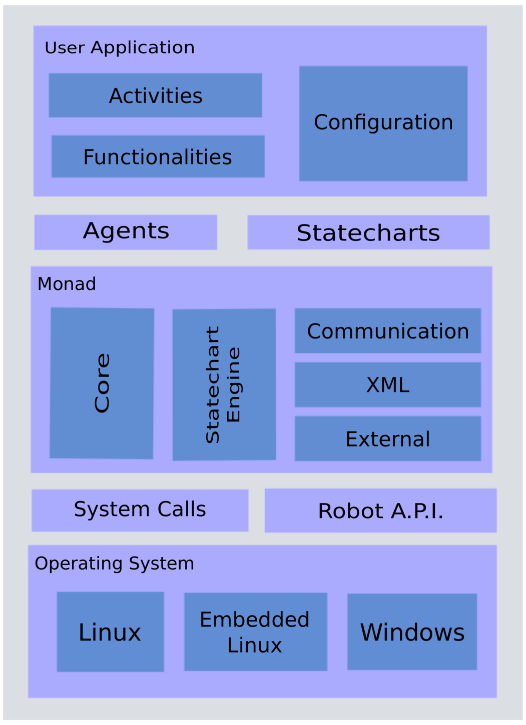


Figure 4.1: Monas Software Architecture Components

agents represents the totality of the robot. Monas's design goals include being a convenient platform for agent developing and debugging.

4.1.2 Agent Decomposition

A major goal of any software architecture is to reach a sufficient level of abstraction in order to organize the source code, so Monas decompose the agent into activities. An activity refers to a simple activity that the agent has the ability to transact. The activity is suppose to be as simple as possible, but is not force to, so providing the developer the necessary structure for organizing the code without compensating its freedom.

To deal with larger activities, Monas provide further decomposition of activities into functionalities. Functionalities are the implementation of what an activity does and each activity must be associated with at least one functionality. Functionalities are source elements that implements a very specific feature and are in the bottom of the hierarchy. Although the latter decomposition step is not optional in the design, it can be omitted in practice when there is no gain by decomposing the underlaying activity, like when the implementation is of the activity functionality pair counts only a few lines of code.

The decomposition makes use of a stronger model in agent creation which is beneficial because it guides the developer to divide the problem and conquer each of the subproblems that arise (following the D&C paradigm), but also improves the code reusability as functionalities and activities can be freely used without modification in as many activities and agents respectively as the user likes. The developer can also trace the execution of the activities and thus the modification and data creation throughout the agent cycle proving a better debugging performance. Another advantage of the activity model is that agents can be modified at runtime by adding and/or removing modules without having to stop the execution of the architecture or even of the agent.

4.1.3 File System Structure

Monas in order to organize the source code, makes use of the filesystem hierarchy structure to separate the source code into divisions relative to the usefulness of each file. The resulting directory tree is presented in Figure 4.2. Each directory is used to hold a certain type of files, so:

- *config* folder contains all the configuration files for both the architecture and the user activities
- *doc* contains the architecture and user-written in-code documentation
- *external* folder holds source code and libraries that are not part of the Monas

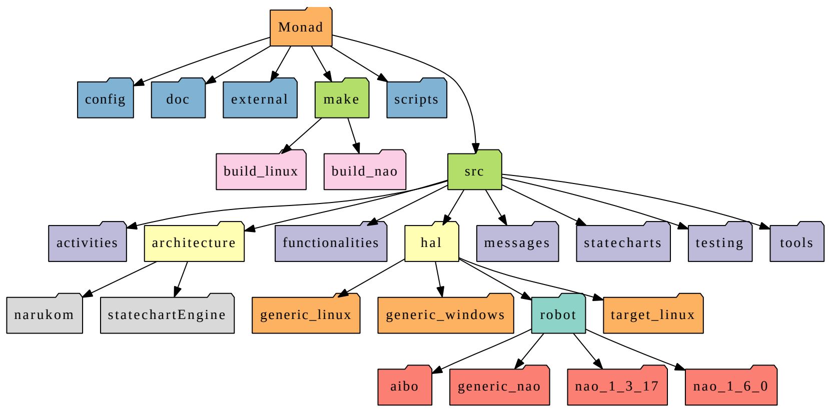


Figure 4.2: Monas's directory tree structure

- *make* directory contain utilities for building the architecture. Currently tow subdirectories can be found under make: *build_linux* and *build_nao* each contains the appropriate configuration for building the architecture for the linux and nao platform respectively.
- *script* is used to store useful user-scripts for setting up, configuring and copying the executables to the robot.
- *src* directory contains the fundamental code for the architecture:
  - *activities* user-defined activities source code
  - *architecture* core-code of the Monas software architecture
  - *functionalities* user-defined functionalities
  - *hal* hardware and platform dependent source code. The folder contains the generic infrastructure for robot support as well as their implementations for specific robots. Also contains code to support building on widely spread platforms, such as Microsoft Windows and Linux.
  - *messages* contains both the *.proto* files as well as the generated *.h* and *.cpp* from the proto compiler. The defines the serializeable classes used for communication.
  - *statecharts* user created statecharts.

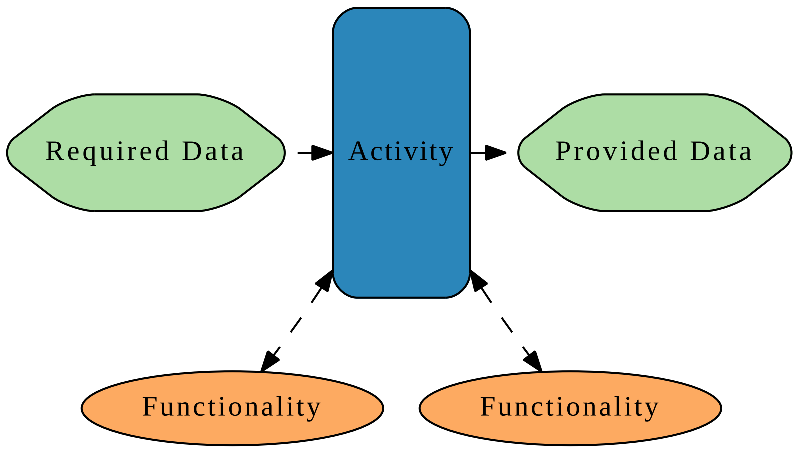


Figure 4.3: Monas's Activity Model

4.1.4 Activities in depth

Activities, as described in section 4.1.2 , are the base units of agents. Activities are used to model the inside of the agent, introducing an abstraction layer that helps the developing process. Activities may vary in size, depending on the developers needs and style, but have a very specific task and accomplish it, they use functionalities. The activity model, as illustrated in Figure 4.3 can be described as an operator on data: requires data before its execution, and provides a data output after execution. The input/output data representations can be formalized as messages, as described in 2.6, so that they can be transmitted without further modification or extensive packaging. Whereas functionalities are closely related with activities in the design model, functionalities are implemented by classes defined by the user, with user created interfaces. Monas does not force any constrains of formalism in functionalities. The developer, in order to use the appropriate functionalities, must include the header files and instantiate the functionalities manually.

The activity interface that is defined by the architecture consists from three abstract functions that the developer must implement:

```
class IExecutable {
    public:
        virtual int Execute()=0;
};

class IActivity : public IExecutable {
    public:
        void Initialize ( Narukom*, Blackboard* );
        virtual void UserInit ()=0;
        virtual std::string GetName ()=0;
    protected:
        Narukom* _com;
        Blackboard* _blk;
};
```

The `Execute()` function, as the name suggests, will be called at runtime when the activity needs to be executed and must contain the code that will accomplish the activity's task. The `UserInit()` is called when the activity is initialized and can be used for communication as well as functionality and class variable initialization. Finally the `GetName()` is used by the architecture to display and manage the loaded activities.

Activities provide a mechanism for inter-agent communication through the Narukom communication system. After the activity instantiation, Monas sets through the `Initialize()` routine the instance of the Narukom and the Blackboard assigned to the underlaying activity. The developer can access Narukom directly through the internal protected variables.

Publishing the activities to the architecture is done by registering the activity into the activity registrar. This is done by creating a temporary variable as is demonstrated in 4.1. The variable is created within an unnamed namespace to avoid namespace pollution. When the variable instantiates an entry containing the activity name and a function pointer to the activity construction will be inserted to the ActivityFactory. The ActivityFactory is implemented as a singleton, so only one instance of the factory may exist. Both ActivityFactory and ActivityRegistrar, which are defined in 4.2, are specializations of the GenericFactory

```
namespace {
        ActivityRegistrar<BehaviorGoalie>::Type temp("BehaviorGoalie");
}
```

Listing 4.1: Registering an activity to the ActivityRegistrar

```
typedef GenericFactory < IActivity , std::string >  ActivityFactory;

template<class T>
struct ActivityRegistrar {
    typedef Registrar<ActivityFactory, IActivity, std::string, T> Type;
};
```

Listing 4.2: Definition of ActivityFactory and ActivityRegistrar

and Registrar respectively. The implementation of the generic factory and its registrar is discussed in section 4.1.6.

Finally, a CMake template is created to ease the building process. The developer has to create a CMakeLists file and set the *ActivityName* and *ActivitySrcs* variables before the inclusion of the activity cmake template file. The process is presented in Listing 4.3. External libraries or other libraries build within the project can be linked normally with the *target_link_libraries* directive. Monas creates the *ActivityBuildType* variable that controls the built of the activity: if is set to *STATIC* the activity will be statically link to the executable, if is set to *SHARED* a dynamically loaded library will be created and if set to *MODULE* the activity will be build as a dynamic library but Monas will load it with a *dlopen* system call. The latter is the most sophisticated approach from the three because does not require the final executable to be linked with the library, making the building process simpler and faster. Monas auto-detects the activities, Listing 4.3, under the activities folder and thus the building system hasn't to be modified each time a new activity is developed.

```
set ( ActivityName Behavior )
set ( ActivitySrcs  Behavior.cpp )
include ( ../activTemplate.cmake )
target_link_libraries ( Behavior NaoQiDep )
```

Listing 4.3: CMake example for building an activity

```
file ( GLOB FilesInDir . * )

foreach ( afile ${FilesInDir} )
    if ( IS_DIRECTORY ${afile} )
        set ( theSubDirs ${theSubDirs}  ${afile}  )
        message ( STATUS
          "Activity ${afile} detected and added to the building tree" )
    endif ( IS_DIRECTORY ${afile} )
endforeach ( afile )

foreach ( subdir ${theSubDirs} )
    add_subdirectory( ${subdir} )
endforeach ( subdir ${theSubDirs} )
```

Listing 4.4: Activity auto-detection

4.1.5 Robot Abstraction & Platform Independence

Monas architecture is designed to be a mobile architecture framework that supports a variety of robotic platforms. The only platform requirements posed by the architecture are the availability of a cpp (cross-) compiler for the on-board computer and the appropriate configuration to accomplish the building.

In order to separate the user's source code from the underlaying platform, Monas introduce an abstraction layer that consists form a set of interfaces. The set is divided into two major sections: one that manages the robots sensors and actuators, and one for managing platform specific and operating system issues. That approach is appropriate because it separates the robot from its operating system and enables the architecture to run on a personal computer while is configured for a specific robotic platform.

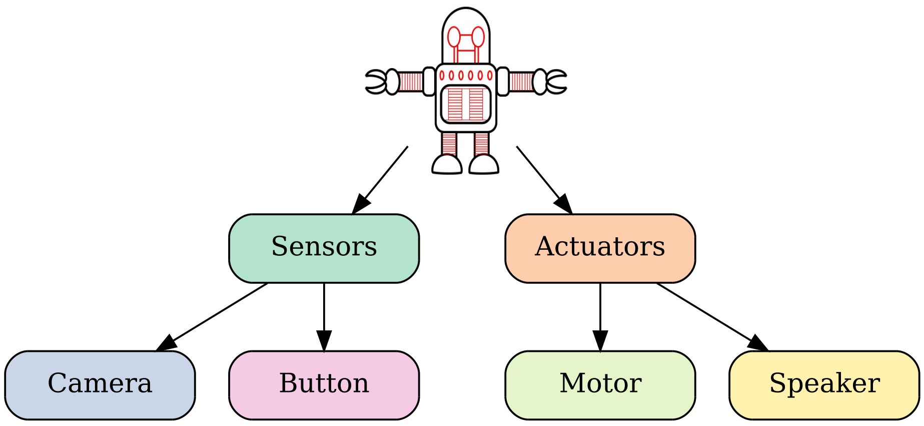
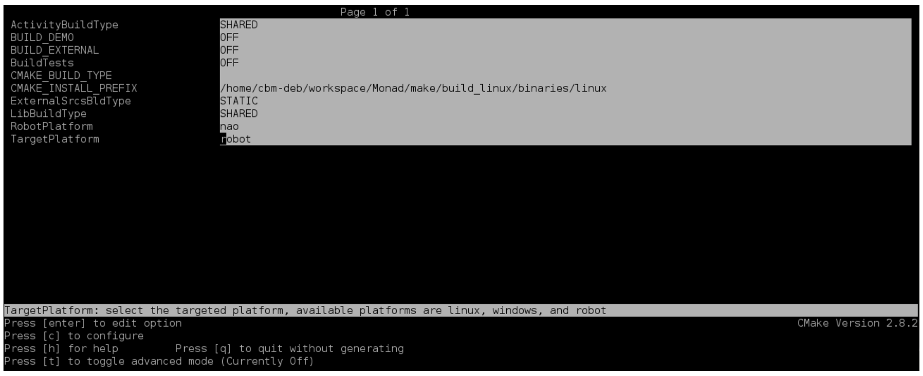


Figure 4.4: Robot Abstraction Model

A robot has multiple means to interact and sense its environment. Robotic actuators, such as the robot motors, affect the robots environment, where as robotic sensors, such as the camera, sense it. The robot abstraction model is illustrated in Figure 4.4. The user can expand the existing interfaces to match its needs. Implementation of the interfaces, which of course depend on the underlaying robot, are located in directory under the robots name (Figure 4.2. A good policy is to control calls to the robot API through these interfaces so that robot sensors and actuators are only accessed through special wrapper calls. If these calls are the only ones with direct access to the robot API, enable the developer to implement, test and debug algorithms that are robot-independent. Direct comparison of these algorithms can be done with ease under different robotic platforms.

Monas is designed to run on different operating systems. To achieve it, system calls and the concurrency model has to modeled. The *`Thread`* and *`Mutex`* interfaces model the concurrency framework whereas the *`SysCall`* interface is used to group and manage the required system calls. Finally the *`Main()`* function, which acts as the application's start point, is also modeled. Monas provide *`Talws`* class which encapsulate the whole architecture, and convenient methods *`Start`* and *`Stop`* direct the execution of the framework. Developer has just to instantiate an object of the class. Thus Monas, except



```
                                          Page 1 of 1
 ActivityBuildType          SHARED
 BUILD_DEMO                 OFF
 BUILD_EXTERNAL             OFF
 BuildTests                 OFF
 CMAKE_BUILD_TYPE
 CMAKE_INSTALL_PREFIX       /home/cbm-deb/workspace/Monad/make/build_linux/binaries/linux
 ExternalSrcsBldType        STATIC
 LibBuildType               SHARED
 RobotPlatform              nao
 TargetPlatform             robot









TargetPlatform: select the targeted platform, available platforms are linux, windows, and robot
Press [enter] to edit option                                                        CMake Version 2.8.2
Press [c] to configure
Press [h] for help          Press [q] to quit without generating
Press [t] to toggle advanced mode (Currently Off)
```

Figure 4.5: CMake building options

from standalone execution, can be integrated with ease in any existing framework.

Building system undertake the task to auto-detect available robots and platforms. Platform dependent options which are required for the correctly building of the software architecture are locate within the interface implementations. Thus, all the necessary platform-dependent files for a specific platform are grouped under the same directory. Figure 4.5 presents the special options that direct the building to the selected platform.

4.1.6 Tools & Utilities

To ease the developing process, Monas provides tools that facilitate the configuration and debugging of agents.

Logger

The `Logger` tool is used to capture useful output from both the agents and the architecture. It separates the information into five main sections according to their significance: `FatalError`, `Error`, `Info`, `ExtraInfo` and `ExtraExtraInfo`. The significance level, determines whether the information

```xml
<?xml version="1.0" ?>
<!-- Configuration for logger module  -->

<MessageLogFile>MonasLog.txt</MessageLogFile>

<!-- MessageLogCerr set to '1' will print logger messages -->
<!-- to standard error, '0' will not -->
<MessageLogCerr>1</MessageLogCerr>
<MessageLogCerrColor>1</MessageLogCerrColor>

<!-- Activities that will reach the LogFile -->
<!-- FatalError and Error directives can't be blocked -->
<!-- Special keyword 'all' will enable all activities -->
<MessageLogFilter>all </MessageLogFilter>

<!-- FatalError =0, Error, Info, ExtraInfo, ExtraExtraInfo -->
<!-- Verbosity level is limited to >= 0 FatalError can't be blocked-->
<LogFileVerbosityLevel>4</LogFileVerbosityLevel>
```

Listing 4.5: Logger Configuration File

will reach the actual output of the logger, or not. The configuration is done through an XML file, as in Listing 4.5, which controls expect from the verbosity level, the filename (and the output path) that the data will be written in, a filter that enables only specific activities to reach the output preventing information pollution, and two special flags: the first one that redirects the output also to the standard error stream, and the second one that colors that output.

Logger also ensures that *FatalError* messages can't be blocked with either selecting a negative verbosity lever or by filtering the activity that is producing them. Additionally, *FatalError*, *Error* messages are flushed to output immediately to assert that can be retrieved even if the architecture crashes.

Using the logger tools is meant to be an easy process. As demonstrated in Listing 4.6, to output an information message the developer calls *WriteMsg* function which is templated and accepts as message any type that implements the streaming operator. The *Logger* tools is instantiated as a Singleton so the

```
class LoggerClass {
    public:
        enum MsgType { FatalError=0 , Error , Info ,
                                         ExtraInfo , ExtraExtraInfo };
        template<class T>
        void WriteMsg ( std::string name, const T& msg, MsgType type );
};

typedef Singleton<LoggerClass> Logger;

Logger::Instance().WriteMsg("Behavior",
        "Error in getting memory proxy", Logger::Error);
```

Listing 4.6: Logger tool interface and usage

```
template<class T>
std::string _toString( T val ) {
    std::ostringstream ost;
    ost << val;
    ost.flush();
    return ost.str();
}
```

Listing 4.7: _toString function implementation

user can access it easily and the architecture asserts that at most one instance of the logger exists.

Using the logger tool can be quite frustrating when the exit message contains a string and a variable because you can not concatenate them. Thus the *_toString* function comes along which accepts any type that can be streamed and returns a string with the result. For the implementation, as is shown in Listing 4.7, we use the stringstream form the standard library in which we stream the input variable.

XML Parsing Library

Monas prefer XML as the configuration language. Although XML is not very human friendly, ensures correctly parsing of the configuration files in any platform. Monas also encourage the developer to use XML for the configuration of the agents, when of course there is no significant overhead. Hence, Monas provide an XML library to read and create XML files. The library comes into two flavors: XMLConfig and XML.

XMLConfig is a simplified version of an XML pareser library and does not support the full schema of the XML. The main simplification the library does is that it manages all XML tags as root tags and thus the tree structure that denotes the XML files does not be preserved. While this sound pretty restrictive when used with configuration files in the architecture is not and the much simpler interface enhances the usability of the library and speed ups the development.

XMLConfig provides two ways to read and modify an XML file. The first way manages tag-element pairs which can be queried and set through the `QueryElement` and `SetElement` methods respectively, whereas the second way manages tag-attributes pairs. Implementing tag-attribute management enable multiple name-variable pairs to be read/written with a single method call. The methods that query and set the variable pairs are overloaded instances of the previous ones. The XMLConfig interface is listed in 4.8.

XMLConfig is a templated library that allow the developer to use any type that can be streamed. Basic types are supported by default and any user defined type can be used by implementing the streaming operators. Both `QueryElement` and `SetElement` methods supports this functionality. When setting up an element, the `isIterative` option controls the creation of new tuples: when false, which is also the default value a new tuple will be created if does not exist and the element value will be store in it. If the tuple exists, the element value will be modified to the new value. When true a new tuple will be created whether a tuple with the same tag exists or not.

```
class XMLConfig {
    public:
        XMLConfig ( const std::string & filename) ;
        bool IsLoadedSuccessfully () const;
        bool SaveConfiguration ();

        template < class T >
        bool SetElement (const std::string& ElName,
                              const T& Value, bool isIterative = false);
        template < class T >
        bool QueryElement (const std::string& ElName, T& Value) const;
        template < class T >
        bool QueryElement (const std::string& ElName,
                                  std::vector<T>& Value) const;
        template < class T >
        bool SetElement (const std::string& ElName,
                const std::map<std::string , T > & Values,
                bool isIterative = false);
        template < class T >
        bool QueryElement (const std::string& ElName,
                std::map< std::string , T > & Values);
        template < class T >
        bool QueryElement (const std::string& ElName,
                std::vector < std::map< std::string , T > >& Values);
};
```

Listing 4.8: XMLConfig class Interfce

In order to store a tag with multiple attributes, XMLConfig implements a *SetElement* overload method that accepts a map. The map uses as key a character string and the value type can be defined by the user. Multiple tag entries can be store and retrieved by using the appropriate vector overloads.

Example 4.9 describes the storage of a pose sequence for the Nao robot, 4.10 the resulting XML file and 4.11 the code that retrieves the sequence. The value type has not been indicated to the library but is automatically detected through the template mechanism. The above method is transparent because poses can be store and retrieved in a generic manner with the sequence that the joints to influent the process.

```
XMLConfig config("test.xml");
std::map<std::string,double> pose1,pose2;

pose1["HeadYaw"]=0.876;
pose1["HeadPitch"]=1.324;
pose2["HeadYaw"]=0.227;
pose1["HeadPitch"]=0.912;

config.SetElement("pose", pose1, true);
config.SetElement("pose", pose2, true);
```

Listing 4.9: Storing of poses in a configuration file

```
<?xml version="1.0" ?>
<pose HeadYaw=0.876 HeadPitch=1.324 />
<pose HeadYaw=0.227 HeadPitch=0.912 />
```

Listing 4.10: The resulting XML file

Monas's XML library, although is does not support the full XML schema, it support much more complicated schemas. This flavor preserves the tree structure and enables searches within specific branches of the tree. To assist the tree traversal, a tree node structure is defined. As listed in 4.12, the node structure contains the tag's name, element and attributes. The developer can then query tags by name and get as a reply a vector of tree nodes. Every node will contain both the element value as well as a map of string-value pairs of the attributes stored in the tag. The query function, which is listed in 4.13, accepts also an XMLNode in the parameters list defining the tree node that will search in to. If no node is passed in the query the search is narrowed to the root of the tree.

The careful reader will note that the XMLNode (List: 4.12) does not have a public accessor or mutator for the `node` field to prevent arbitrary access to the tree structure. It encapsulates the tree information and can be used only in combination with the query. The only assertion made in the implementation of the library is that all attributes inside a tag are of the same type.

```
XMLConfig config("test.xml");
std::vector<std::map<std::string,double> > poses;
bool found = config.QueryElement("pose", poses);
```

Listing 4.11: Reading the poses from the configuration file

```
template<class TxtType, class AttrType, class Key = std::string>
class XMLNode {
    public:
        std::string name;
        TxtType value;
        std::map<Key,AttrType> attrb;
        XMLNode(std::string name) : name(name), node(0);
    private:
        const TiXmlNode * node;
        XMLNode( std::string name, const TiXmlNode * node );
        friend class XML;
};
```

Listing 4.12: XMLNode class

Both library flavors do not parse the XML file by themselves but instead they use TinyXML parser. The libraries act as an additional abstraction layer that maintain a constant interface which is not subject to variations by changes in the TinyXML's API. Additionally the parser change is possible with minimum changes in the source code. The newly defined user interface, especially for the XMLConfig library, is much easier to use than a full parsers interface, enabling the developer to prefer XML as the configuration language.

Abstract Factories

To deal with the problem of activity instantiation given the activity name in string, Monas implements the abstract factory pattern. The generic factory class, Listing 4.14, maintains an associative container, currently set to map, to keep product identifications and pointers to object construction entries that will allow later

```cpp
class XML {
    public:
        XML ( const std::string & filename);
        template <class TxtType, class AttrType, class Key >
        std::vector<XMLNode<TxtType, AttrType, Key> > QueryElement (
                const std::string & ElName,
                XMLNode<TxtType, AttrType, Key> * pNode = NULL) const;
};
```

Listing 4.13: XML Query Class

to instantiate the product. Currently products that take none, one or two arguments at their constructor are supported but can be extended to almost any number of arguments by creating additional *CreateObject* methods.

In order to ease the registration of a product a *Registrar* was created. The registrar is responsible to create the appropriate functions that create a product and then register the product and the created function to the factory. The creation of the functions exploits the fact that given a template class the compiler implements only the functions that are called in the source code and only for the specific type. *Registrar* creates the overloaded function *NewProductFunc* that accepts up to three parameters and returns a new product instance. The compiler then selects the appropriate function given the number of parameters. Extension to support more parameters can be easily done.

Monas make use of the abstract factory pattern in the cases of activity and thread instantiation. Abstract factories are easy to used as demonstrated in Listing 4.2 and 4.1 for using with activities.

Singleton

As the *Singleton* pattern is used in the architecture, an abstract singleton implementation had to be created. The Singleton class has private constructor

```
template <
  class Product,
  class IdType,
  class ProductCreator = Product* (*)(),
  class T1 = bool,
  class T2 = bool,
  class ErrorPolicy =PrintErrAndExitPolicy >
class GenericFactory : public ErrorPolicy {
    public:
        bool Register(const IdType& id, ProductCreator cr);
        Product* CreateObject(const IdType& id );
        Product* CreateObject(const IdType& id, T1 p1 );
        Product* CreateObject(const IdType& id, T1 p1, T2 p2);
    private:
        typedef std::map<IdType, ProductCreator> Id2TypeMap;
        Id2TypeMap assoc;
};
```

Listing 4.14: Abstract Factory Implementation

and destructor and unimplemented copy constructor and assignment operator so that singleton objects can not created outside the singleton class and can not be copied. The class is templated so any class can act as a singleton by using a typedef directive. The instance can be accessed through the `Instance` method which returns a reference to the statically created object. Listing 4.15 indicates the `Singleton` implementation and demonstrates its usage. The singleton pattern is used in the architecture for the logger and the abstract factory implementations.

Stopwatches

Stopwatches allow measuring the execution time on a code segment. The users should instantiate a `StopWatch` object and select a policy for the time measurement. Default policy calculates the time by using exponential moving average, so instead for the specific time interval, the average is returned.

```
template< class T>
class Singleton : public T {
    public:
        static T& Instance() {
            static T t;
            return t;
        }
    private:
        Singleton();
        Singleton(const Singleton&);
        Singleton& operator=(const Singleton&);
        ~Singleton();

};

typedef Singleton<aClass> aSingletonClass;
```

Listing 4.15: Singleton implementation and usage

```
template<class AvgPol = StatMovingAverage>
class StopWatch : public AvgPol {
    public:
        void StartTiming ();
        double StopTiming ();
};
```

Listing 4.16: StopWatch class interface

Additionally, the policy provides the average variance also calculated by exponential moving average algorithm. The interface of the stopwatch class is listed in 4.16. The policy based design enables implementations of new policies to be easily added on the stopwatch; new policies may define a new interface that is available only on objects that uses the policy, without interfering with the rest.

```
<agent IsRealTime=1 Priority=1 ThreadFrequency=10 StatsCycle=15>
  <name>Motion</name>
  <activity>Vision</activity>
  <activity>Behavior</activity>
</agent>
```

Listing 4.17: XML agent definition

4.2 Agent Management

To control the agent creation a primitive agent management system was implemented. Agents are defined in an XML file, Listing 4.17, and instantiate at runtime. Each agent runs at its own thread with its activities executed sequentially. The proposed approach has the advantage that is very easy to understand and use efficiently. Managing agents from an XML file that is required at the start of the architecture, gives the developer the freedom to change the components of the agent and create new agents without the necessity to recompile the source code. Monas also supports to start and terminate agents on the fly, by sending the appropriate network message. Agent modification is also supported over the network but the interface is not implemented yet.

Monas parse the XML configuration file and instantiate the agent. As shown in Listing 4.18, the agent derives from the Thread class which deduce that the agent will run at its own thread of execution. The set of the activities passed to the agent through its constructor in a form of a string vector and then the agent instantiates the activities with appropriate calls to the activity factory, Listing 4.19. As presented in Listing 4.17, the agents thread can be configured through the XML file. The *IsRealTime* and *Priority* attributes control the kernel scheduling of the underlaying thread and are compatible only with pthreads. The *ThreadFrequency* attribute controls the number of execution cycles in a second (or the Hertz). The execution cycle starts when the first activity in the activities list starts its execution and ends when the last activity has finished. The agent then sleeps for *1/ThreadFrequency − ExecInterval* seconds before the start of the next cycle. If the resulted time is negative, a warning

```
class Agent : public Thread {
    public:
        Agent ( std::string name, AgentConfig cfg,
                Narukom* com, std::vector<std::string> activities )
        virtual ~Agent ();
        int Execute ();
```

Listing 4.18: The agent class interface

```
for ( ActivityNameList::const_iterator it = activities.begin();
         it != activities.end(); it++ )
  _activities.push_back(
          ActivityFactory::Instance()->CreateObject( (*it) ));
```

Listing 4.19: Agent activity creation

message is print in the logger and the execution continues immediately to the next cycle. Developer has the ability to time schedule the activities controlling the ThreadFrequency and, indirectly, control the cpu load. The agent model is thus suitable for any type of agents, from reactive agents to complicated computer intensive as well as any combination of them.

Debugging procedure is also boosted by this agent management system, as the developer can isolate the components of the agent one by one until it creates a minimum set of activities that reproduce the error so that the search for it is narrowed. Furthermore, as it can been noticed in Listing 4.17, the developer can print statistics from the time scheduling of each agent as well as of each activity inside the agent. The *StatsCycle* attribute controls the frequency the statistics will be printed. Time statistics are smoothed using the exponentially weighted moving average for the mean time of execution for both agents and agent's activities but the variance is also been calculated to help the developer to spot inconsistencies in the expected execution time.

4.3 ASEME Methodology

In our quest for a principled approach to designing complex elaborate agents, we found that ASEME 2.5 and AMOLA offer a number of advantages compared to other related methodologies. ASEME is a convenient methodology for agent developing and enhances Monas to a complete architecture which supports all phases in the physical agent developing process, from requirement analysis to agent design. The final step of the realization of the designed agent is to implement and configure the agent for the specific hardware platform.

ASEME, is model driven development (MDD) process which guides the developer from gathering requirements to an implementation Platform-Specific Model (PSM). These platform-specific models, can be automatically or manually transformed to source code. ASEME produces a platform-independent model (PIM) as the outcome of the design phase that includes the *Intra Agent Control (IAC)* and the *Inter Agent Control (EAC)*, which are based on the formalism of statecharts. Thus, we need a run-time statechart execution engine for instantiating these models on the robotic platform.

4.3.1 Yet Another Statechart Engine (YASE)

YASE statechart model details and comparison with existed models

The statechart model that this engine supports has differences from the common ones found in the literature([HN96, HK04, Gro05]). Compared to the most popular platforms, i.e. the *Statemate tool* [HN96], *UML 2.0* [Gro05] and *Rhapsody tool* [HK04], the notation and schematics defined and used by *YASE* are closer to the *Rhapsody* tool. The differences are in most of the cases quite simple, but their effects can dramatically change the execution behaviour and lead to unexpected results.

First of all, before analyzing the differences, the *transition expression* grammar must be defined. In a transition expression the user should be able to, optionally, define *events* and *conditions*, as well as multiple actions if desired. Additionally, the grammar must describe the access of variables in order to create a unified scheme. At *YASE*, the variable model is similar to an ontology, and variables can be described by their message type (a structure similar to the object-oriented class) and the message's element (a class public member). If the variable originates from an other capability, but within the same host, the capabilities name has to be specified before the message and element names. Finally, to access a variable sent by another host, the modeler has to specifies the agent's name before the rest of the expression. In *YASE*, there are two special action directives, *"process_messages"* and *"StartTimeout"*. The former applies an update to the communication system and the latter, as the name suggests, starts a timeout. When the timeout expires, a timeout event is send to the engine. The transition expressions are defined in *EBNF* format (based on Russel) and are depicted in Figure 4.6).

YASE adopts the strategy of defining *START-states* and *END-states* as in the *UML 2.0* specification. While the *START-state* substitutes the default transition scheme of *Rhapsody*, the *END-state* represents a state with no outgoing transitions. The latter is modeled as a simple state, rather than a *psedostate*, as the transition algorithm, must be able to "stop" when in this state. Additionally, more than one transitions, with conditions (guards) in their expressions, can originate from a *START-state*, as opposed to *UML 2.0*. The limitations that are posed in this case are that all the outgoing transitions targets must be inside the composite state and that at least one, not to have a condition.

The engine gives the freedom to the developer, to decide if he wants to use *condition-states* or multiple transitions connecting the source states to the all targets states. Both cases will lead to the same active configuration with the latter having a performance penalty on a high number of outgoings transitions. When a transitions is taken, conditions are evaluated before the any action's execute, a structure similar to the static choice of the *UML* specification.

Unlike *STATEMATE* and *UML*, the engine is following *Rhapsody* approach and does not support neither conjunction, disjunction or negation of events. The

```
transitionExpression = [ event ] [ ”[” condition ”]” ] [ /action ]
event = string
condition = variable compOp ( variable | value )
              | condition logicOp condition | ”(”condition”)”
              | ”not” ”(”condition”)”
action = variable ”=” ( variable | value )
           | action connectiveOp action | ”process_messages”
           | ”StartTimeout(” variable ”,” time ”)”
compOp = ”<” | ”<=” | ”>” | ”>=” | ”==” | ”!=”
logicOp = ”&&” | ”||”
connectiveOp = ”;”
variable = host ”.” cap_name ”.” message ”.” element
             | cap_name ”.” message ”.” element
             | message ”.” element
time = digit_list ”ms”
host = string
process = string
message = string
element = string
string = letter_or_digit | letter_or_digit string
letter_or_digit = letter | digit
letter = ”a” | ”b” | ”c” | ”d” | ”e” | ...
digit_list = digit | digit digit_list
digit = ”0” | ”1” | ”2” | ”3” | ...
```

Figure 4.6: The transition expressions grammar in EBNF.

approach appears to have functionality loss comparable to the other implementations, but thats not the case: disjunction functionality can be implemented by creating a transition for each event on the disjunction while the negation, given the universe of events, can be transformed into a disjunction, adding also a transition with no event. Event conjunction not be implemented, as the engine adheres the concept of run-to-completion (RTC). RTC is interpreted to ensure the correct dispassion of events. Each event processing must have been finished before the following event is dispatched. The *RTC* concept also applies to the *UML* implementation, and thus event conjunction is not supported neither.

As we analysed in Section 2.4, if more than one transitions can be executed during a step, are considered in conflict, if their effects resume to different active configurations. *YASE* follows the object-oriented design approach and, thus, gives priority to the transition with the lowest scope. The approach is followed by both *UML 2.0* and *Rhapsody tool* but not by *Stetemate*. Executing a transition, either the transition itself or the transition actions are not consider to take a significant amount of time, but it take only a fraction of time. The time can be approximated as a fixed-execution time or as zero- depending on the semantics that the modeler defines.

On a transition between two states, the transition expression may contains none, one or multiple actions. Again, YASE, follows *UML 2.0* and *Rhapsody* so the actions are executed in the defined sequence, instead of the parallel execution that is supported by *Stetemate*. This case is very important, because the difference is on the semantics and statecharts in different formalisms, while are correctly designed, their execution may lead to unexpected results.

Furthermore, it should be noted that YASE adheres the “step” concept that was introduced by *Rhapsody tool*. As a result, in the scenario in which a transition’s target is a composite state, then the action of the underlaying transition will be executed before the default transition of the composite state. This lead to a conflict with *UML*, as if outgoing transitions from the *START-state* contain conditions, then the data that the conditions will be evaluated with, may have change by the action which “targets” the composite state.

In Table 4.1, there is a comparison between *YASE* and the existing models of statecharts. At the first column, we present the *UML 2.0* mode and, as *UML* has been the “de facto” standard in software modelling, the rest models are compared with it. If the first part of the table, we cover differences in the syntax of the model where as in the second on the semantics.

Table 4.1: Supporting Attributes and comparison between *UML, Rhapsody and YASE* statechart formalisms. Table has been inspired from [CD07]

Construct/Concept	UML	Class.	Rhapsody	YASE	Boost
Syntax					
States					
entry/exit actions	●	⊙	●	●	●
do-activity	●	⊙	⊙	●	⊗
deferred events	●	⊗	⊗	⊗	●
Pseudostates					
initial	●	●	●	●	⊙
final	●	●	●	●	⊗
fork	●	⊙	⊙	⊗	⊗
join	●	⊙	⊙	⊗	⊗
shallow history	●	⊙	⊗	⊗	●
deep history	●	⊙	⊙	⊗	●
junction (static)	●	●	⊙	⊗	⊙
conditional (static)	N/A	●	●	●	⊗
choice (dynamic)	●	⊗	⊗	⊗	⊗
Transitions					
event trigger	●	⊙	⊙	⊙	⊙
action (behaviour)	●	⊙	●	●	⊙
completion	●	⊗	⊗	●	⊙
event disjunction	●	●	⊗	⊗	⊗
event conjunction	⊗	●	⊗	⊗	⊗
event negation	⊗	●	⊗	⊗	⊗
compound trans.	⊗	●	⊗	●	⊗
null transitions	●	●	●	●	⊗
Semantics					
simultan. events	N/A	●	N/A	N/A	⊗
seq. action exec.	●	⊗	●	●	⊙
bottom-up priority	●	⊗	●	●	●
Open-source platform	N/A	⊗	⊗	●	●
Multi-Threaded support	N/A	⊗	⊗	●	⊗

Legend for Table 4.1

Symbol	Description
●	The concept is supported by the formalism
⊙	The concept has considerable differences from *UML 2.0*
⊗	The concept is not supported by the formalism
N/A	not applicable

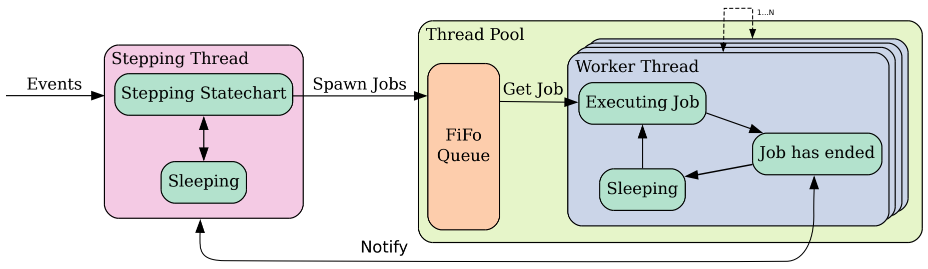


Figure 4.7: Statechart's engine architectural model

YASE Implementation

Statecharts major difference from finite state machines and its derivatives is that they natively support concurrency. When on a finite state machine only one state can be active, whereas on statecharts multiple states can, modelling that way orthogonality which occurs anyway in agent design. To support that in the engine itself, without introducing loss of the system responsiveness, we split the execution of the engine into two parts. The first part is responsible for stepping the engine and is executed in its own thread, whereas the second one is responsible for executing all the activities that are active at the time, implemented as a thread pool. The implementation model of the statechart engine is illustrated in Figure 4.7. The implementation enables the developer to specify the maximum number of activities that can be active simultaneously, managing the CPU load and taking advantage of multi-cores CPU's.

The tread pool initializes a specific amount of threads, currently set to twelve, and a FiFo queue. When a job arrived at the pool, enqueues and a signal wakes up a thread. The thread dequeues then and execute the job. When the job is finished it will try to dequeue an other job and execute it. If there are no other jobs in the queue, sleeps in a condition variable. Other scheduling scheme can be used, such as a priority queue, but with the current design all jobs which are actually agent activities run with the same privileges and priority so the needed information for prioritizing the jobs is not present. A LiFo scheme would be really bad choice as it would introduce random latency and performance decrease on a system load.

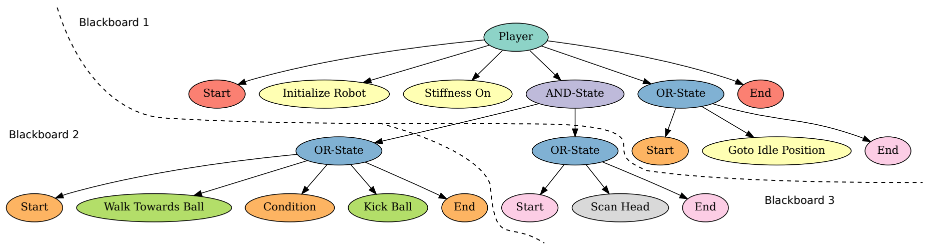


Figure 4.8: Blackboard scope in Statechart Engine

Monas's statechart engine, incorporates the Narukom communication system for inter-activity and inter-statechart communication, giving the ability to every activity, action and transition to direct access a Narukom instance, which is unique within the underlaying statechart, and to a blackboard instance. Thus, activities can communicate, actions can interact with the environment and transitions evaluate their conditions using the publish-subscribe system as described in 2.6. The blackboard instance is not one for every statechart instantiation but its is scoped. The scope include all the states between the lowest level OR-state that is a common ancestor of both the source and target states. The concept is illustrated at Figure 4.8. As a result, activities that live in a different region of an AND-state will not have the same data at a specific time, this does also apply to the evaluation of transitions expressions in which the same condition may evaluate to true in a region whereas in an other one to false.The communication model solve the synchronisation issues that arise, such as the consumer-producer problem which occurs when an activity is executed more times than an other activity which uses data provided by the first, as well as the starvation problem when multiple threads try to lock the same mutex simultaneously, that would have been occurred if have used a shared-memory model for the communication.

Monas statechart engine implementation has slightly different semantics from the Harrel's *Rhapsody* tool [HK04]. Beyond the OR-state, AND-state and BASIC-state, it defines the START-state and END-state, as in UML [Gro05]. Theses states are introduced to formalize the default transition inside an OR-state and to indicate that the inner state has reach its end of execution respectively. These states are useful because they ease not only the development of the engine but also the visual representation of the statechart. Also transition expressions can, of course, be included in their outgoing and incoming transitions making further control of the entrance and exit of an OR-stage formalized. As illustrated in the class diagram, Figure **??**, every state can have an entry and an exit action. Actions deffer from activities as they do not consider to consume any, significant, computational time. Both actions are optional and will executed in the activation and deactivation of the state. An activity must be assigned to the BASIC-state. On the activation of the state, the activity is enqueued to the thread pool and is then executed, according to the thread pool internal algorithm, on a separate thread. The activity is not executed endlessly as long as the state is active but it only runs once. As long as the activity is running, a shared mutex is lock so the states which share the same blackboard can not step to ensure the correct execution of the statechart. Transition segments support transition expressions which is represented visually as *`e[c]/a`* with *`'e'`* denotes the event, *`'c'`* the condition and *`'a'`* the action. The transition expressions controls the execution of the transition: if the event match and the condition evaluate to true only then the action is executed. In the transition algorithm the event is disambiguated by the cpp *`typeid`* which indicates the class type of an object at runtime. Transition segments can orientate and/or reach the states that are described above or special states called connectors, forming compound transitions. A compound transition can be either executed as a hole, which means that every transition segment that participate must be executed, or not executed at all. From the connectors introduced in Rhapsody only the condition connector is implemented. Other connector can be implemented with ease as the engine is written to be expandable. Transition segments are template classes with arguments the source and the destination type. The type, which can be done either a normal state or the condition connector, is used by the compiler to select the appropriate transition algorithm at compile time.

The stepping algorithm for the statechart engine is distributed among the state and transition classed that constitute the statechart. The execution starts form the *`Statechart::Execute()`* method. While on running mode, it tries to step the statechart until no transition can be taken. As as step is consider the execution of at list one transition segment. More transition executions

Algorithm 4.1 Statechart's main thread execution algorithm

```
loop
  while Step() = true do
    do_nothing
  end while
  mutex_lock()
  if notified = true then
    notified ← false
    mutex_unlock()
    continue
  end if
  condition_var.wait()
end loop
```

can occur in a single step if, and only if, are in different AND-state regions or are parts of a compound transition. When further stepping is not permitted, the execution thread sleeps in a condition variable. The algorithm of the `Statechart::Execute()` method is shown in Algorithm 4.1.

The stepping algorithm for *BASIC-state*, *START-state* and *END-state* is the same and is implemented in the State base class, so the sub classed derive the method. The algorithm, which is presented in 4.2, tries to execute transitions that have been added to the state, by the addition order, until it finds one that can be executed, so that the method returns true. The method returns false when an activity, in the region that the states belong, is executing or if none of the states transitions can be executed. In an *OR-state*, the stepping algorithm tries to step the active substate and if fails then tries the *OR-state* itself. Again if an activity is running on the region the method returns false immediately. The algorithm is illustrated in Listing 4.3. Finally, at an *AND-state*, tries to step every region in the state. If succeed to step at least one, returns true, else is at least one of the regions was running before the it tries to step, return false. In the case that neither a step could be taken on a region, nor any activities was running on the regions, the algorithm tries to step the AND-state itself by calling the, base class, Step method. The algorithm for stepping an AND-state is presented in 4.4. It must be mentioned that the stepping algorithm selects the first transition that can be followed, so if two transitions can the result is undefined. At the current

Algorithm 4.2 Generic stepping algorithm for BASIC-, START-, END-state

```
if isRunning then
  return false
end if
for tr in transisions do
  if tr->Execute()=true then
    return true
  end if
end for
return false
```

Algorithm 4.3 Stepping algorithm for OR-state

```
if isRunning then
  return false
end if
if activeState->Step() = true then
  return true
else
  return State::Step()
end if
```

implementation the evaluation of the transitions, thus the selection, matches the order that the transitions were added to the state.

On the addition of a new transition the transition must be initialized and the deactivation and activation lists must be drawn up. These lists specifies the order that the states will deactivate and activate respectively when the transition executes. To implement the functionality the algorithm starts from the state that the transition originates and completes a list of every ancestor until the root state is found. The same process also applies to the destination state but the list is created in a reverse order. Then we search the first list for a state that exists in the second list. The state that is found is the lowest common ancestor of the originate and destination states. The deactivation list will contain all ancestors from the origin to that state and the activation list will contain all ancestors from the destination state but in the reverse order. Thus the implementation matches the transition algorithm proposed in the *Rhapsody* tool [HK04]. Pseudo-code is presented in Listing 4.5.

Algorithm 4.4 Stepping algorithm for AND-state

```
 isRunning ← false
for state in substates do
  if state->isRunning() = true then
    isRunning = true
  end if
end for
 stepTaken ← false
for state in substates do
  if state->Step() = true then
    stepTaken = true
  end if
end for
if stepTaken = true then
  return true
else
  if isRunning = true then
    return false
  else
    return State::Step()
  end if
end if
```

Algorithm 4.5 Transition initialization algorithm

```
List srcAncestors
State parent = src_node
repeat
  parent = parent.GetParent()
  srcAncestors.push_back(parent)
until parent $\neq$ rootState
List trgAncestors
parent = trg_node
repeat
  parent = parent.GetParent()
  trgAncestors.push_front(parent)
until parent $\neq$ rootState
for i in srcAncestors do
  for j in trgAncestors do
    if i = j then
      deactivationLst.assign(srcAncestors.begin(),i)
      activationLst.assign(++j,trgAncestors.end())
      return
    end if
  end for
end for
```

Algorithm 4.6 Transition detection of execution

```
Require: Event e
  if hasEvent then
    if $e \neq 0 \vee typeid(e) \neq typeid(event)$ then
      return false
    end if
  else
    if $e \neq 0$ then
      return false
    end if
  end if
  if hasCondition then
    if $condition.eval() = false$ then
      return false
    end if
  end if
  return true
```

When the `Execute` method is called, it tries to execute the transition segment and if succeeds return true, otherwise false. The transition algorithm depends directly to the type of the destination state. As a first step on the algorithm, despite the destination type, the segment checks if it can executes: if the transition expression contains an event then the event must match the event passed to the execute function and if contains a condition, the condition must evaluate to true. The algorithm is presented in 4.6. When the destination state is not a *condition connector* and the transition segment can execute, first the source state is deactivated which is followed by the states in the deactivation list. Then the transition action is executed if exists, states in the activation list and the destination state activated. The algorithm is presented in 4.7. When the destination state is a *condition* connector, first a valid path of transition segments that can be executed is found and then each of the participating segments executes.

While *YASE* was designed and developed with the current implementation of ASEME, the engine can be used as a stand-alone library. The required code that the user must developed has been keep as simple as possible, which speeds-up development, improve code robustness and ease the maintenance process. *YASE* fully supports transitions in any applicable scenario, but does

Algorithm 4.7 Transition execution algorithm

```
if CanExecute(e) = false then
  return false
end if
src_node.deactivate()
for i in deactivationLst do
  i.deactivate()
end for
if hasAction then
  action.Execute()
end if
for i in activationLst do
  i.activate()
end for
trg_node.activate()
```

not provide the user with *junction, fork, joint* and *history* pseudostates. These pseudostates, while are not currently supported, *YASE's* extensible design allow the developer to implemented them depending on it's needs.

4.3.2 Transforming ASEME Models for YASE

The final step on the ASEME process, is the transformation of the resulting model from the design phase to the Monas architecture. In the last phase of the design, the modeler merges the *Inter-Agent Control (EAC)* into the *Intra-Agent Control (IAC)*, to facilitate a complete model, which describe the entity of the agent as whole. The final model, which is still referred as IAC, is platform independent and based on the formalism of statecharts, thus to implement it, must be transformed into source code compatible with *YASE*, Monas's statechart engine.

The model utilize XML Metadata Interchange (XMI) format as the representation for its data. Hence, a model-to-text (M2T) transformation is needed to transliterate the model into source code. As IAC, is being defined through the use of statecharts and having the statechart engine designed to fulfill ASEME requirements, there is a, close to, injective (one-to-one) relationship between

Table 4.2: IAC to Statechart Engine Node Transformations

IAC Node Types		
Tag	Type Attrb.	C++ Source Code
IAC:root	OR	Statechart «label»("«name»", com);
IAC:Node	OR	OrState «label»("«name»", «Parent(label)»);
IAC:Node	AND	AndState «label»("«name»", «Parent(label)»);
IAC:Node	BASIC	IActivity Activ«name» = CreateActiv(«name»); BasicState «label»("«name»", «Parent(label)», Activ«name»);
IAC:Node	START	StartState «label»("«name»", «Parent(label)»);
IAC:Node	END	EndState «label»("«name»", «Parent(label)»);
IAC:Node	CONDITION	ConditionConnector «label»("«name»", «Parent(label)»);

the model and the engine. In order to achieve the transformation, XPand language 2.7.3, which is offered by Eclipse Modeling Project[1] was mobilized to transform the IAC XMI model to C++ source code.

Transforming the IAC models is a trivial process, and basically depends from the `IAC::Node` type. For each type, Table 4.2 illustrates the automatically generated source code. The transformation process starts by searching for the root node in the XMI file and it transforms it to an instance of the `Statechart` class. Then, every other node is expanded to source code by instantiating the appropriate class of the statechart engine according to the node's type. Node expansion follows and preserves the hierarchy of the model, so that all the ancestor nodes to be instantiated before the current node.

Afterwards, transitions are expanded. The expansion depends from the source and target types. Special attention is given when either the source or the target type is a *condition connector*. The transformation into source code is given in Table 4.3. Transition expressions are also evaluated in each transition node and if any combination of event, condition or action is detected then the expansion stalls until the expansion of the transition expression is completed first.

[1]The Eclipse Modeling Project provides a unified set of modeling frameworks, tooling, and standards implementations [BBM03].

Table 4.3: IAC to Statechart Engine Transition Transformations. The *Src* and *Dst* noted with "*" can be any type of *state* or *pseudostate* except the *condition connector*.

IAC:Transition Types		
Src. Type	Dst. Type	C++ Source Code
*	Condition	TransitionSegment<State,ConditionConnector>(«source.label»,«target.label»);
Condition	*	TransitionSegment<ConditionConnector,State>(«source.label»,«target.label»);
Condition	Condition	TransitionSegment< ConditionConnector,ConditionConnector>(«source.label»,«target.label»);
*	*	TransitionSegment<State,State>(«source.label»,«target.label»);

Finally, to complete the transition to Monas architecture, appropriate buildings files must be generated to enable seamless compilation of the statechart. Thus, `CMakelists.txt` file is created, a file compatible with *CMAKE* building system. The, newly designed, statechart is now ready for usage and should be placed under `Statecharts` directory, in Monas's filesystem. To execute it, the user should modify the `config/Agents.xml` file appropriately.

As XPand closely collaborates with XText, it allows of Java helper functions to be implemented. These functions provide functionality to detect and analyze transition expression as well as modification on node names using Java native libraries for string manipulation. This is extremely useful as the *name* and *label* attributes in *IAC:Nodes* contain invalid characters in the *C++* namespace, such as dots ('.'). With the help of theses functions we can transform them into dashes ('-') or underscores ('_') by a simple call.

Chapter 5

Results

In this chapter, our approach is going to be tested and evaluated using various methods as an attempt to provide objective results. As the subject of this thesis is more about the quality of the source code that enables to be written rather than the quantity, the problem of measuring the efficiency and the added value arises. To begin with, the evaluation procedure is presented in the next section, followed by the experiments that took placed and finishing with the results.

5.1 Testing and Evaluation Procedure

To evaluate all the proposed solutions, we develop agents in the RoboCup field. RoboCup, as being a difficult robotic environment, requires complex, deliberate agents to be developed. Additionally the RoboCup competition represents a challenging multi-agent environment, whereby individual robot skills alone cannot lead to team success. Furthermore, the real-time constraints on agent operation, imposed by the robotic hardware dependencies, makes the task more difficult and inappropriate for many agent-oriented software engineering approaches, as it will be insufficient in designing effective robot teams. Finally, RoboCup is a very competitive environment, where most researchers benchmark their work in the real-world problems that it poses and it is not uncommon for calibration and fine-tuning on both algorithms, agents and robots up to the last-minute before the game start. Hence, RoboCup is an ideal field for the Monas's thorough testing and evaluation procedure as it requires not only complex agent design and advanced robotic control but also as user friendliness and reduced developing times.

The evaluation will be deployed in the RoboCup's standard platform league as well as the, currently unofficial, Webots simulation league using Aldebaran's Nao robot. As both SPL and Webots are soccer leagues, enables testing and evaluation on Monas's platform independence as well. Two agents were decided to be developed, an attacker and a goalie player. The agents will be implemented using four different approaches:

- *Direct on NaoQi*, Aldebaran's middleware which provides, beside the API for controlling the robot, a platform for modular developing and a thread-safe mechanism for communication.
- *Monas's Agents*, which directs to a divide and conquer developing approach. The developer creates functionalities and activities that can be used on the robot and then configures the execution order in different threads of execution with a convenient XML file.
- *Agent System Engineering Methodology*, a complete methodology that first analyze and then through a series of model transformations aid the developer in the design process.

Each approach is evaluated through the use of various metrics that try to cover the majority of aspects of Monas software architecture. As the quality of the software can't be quantified other metrics must be used in order to approach, as objectively as possible, the evaluation measurement. The metrics that are used, together with a brief explanation and approval of its usage, are the following:

- *Source Lines of Code*, whereas is an controversial metric, it applicable for Monas's evaluation as every approach yields to the same functionality, using the same developing language, and using the same software paradigm. Additionally, it counts the source code created by the same individual so that source code style and comprehensiveness remains the same. This metric is concrete and very easy to measure both manually or even with an automated process.

- *Total Developing Time* to achieve exactly, or equal if not applicable, functionality among each approach. The total developing time includes the time needed to complete the developing process starting from scratch and includes time for as many re-design phases needed. Time spend for debugging purposes is excluded.

- *Number Global State Variables* used to describe analytically the state of each player. These variables are controlled by the user on NaoQi and Monas's agents implementation in contrast to the transparent, automated management that achieved with the Statechart based implementations. It is design to quantify the increasing difficulty on complex agent creation when the user does not use a state engine to implement the agent. The metric is appropriate because the same agent is created with various implementations whereas it would be inapplicable otherwise.

- *Code Cohesion* determines how strongly-related is the functionality expressed by a part of source code. There are several rankings that describe the level of cohesion of the source code.

- *Code Coupling* is the degree to which each program module relies on each one of the other modules. High code coupling main disadvantage is that a change in one module usually leads to a ripple-effect of changes in the dependent modules and thus is considered a drawback in the design.

- *Run Time Performance* determines the system load when the agent runs. As the system remains the same (Nao robot) and each agent performs the same task it is an objective metric for evaluation of the aforementioned agent developing approaches.

- *Debugging Time and Number of Bugs*, is the time consumed for debugging the agent. Combined with the total number of bugs that found can provide a convenient metric of how easy or difficult the debugging process is.

5.2 Agent Description

In order to evaluate our approach two RoboCup agents were developed: an attacker and a goalkeeper. The agents are kept simple enough to fulfill a minimum of a RoboCup player as the thesis propose an architectural approach to the problem and not a complete player. The attacker, is able to communicate with the game controller and, when the game state is set to *PLAYING* , approach the ball and kick it. Goalkeeper accordingly must be able to defend its goalpost by falling or diving when the ball is approaching the goalpost. Decision between falling or diving depends from the approaching angle and speed. Agent behaviour can be described in an algorithmic form as in Algorithm 5.1, which refer the attacker execution plan and Algorithm 5.2, for the goalkeeper.

In the design process, a set of, already developed, source code components that provide common functionality is integrated into the agents to simplify the developing process. Namely the components are:

- *RobotController*, implements the state machine which provides the game state to the rest of the platform, as it is being documented at the latest RoboCup SPL rules [Rob10]. The component achieves its functionality by capturing torso button presses as well as listening to the Game Controller special software, provided by SPL the organizing committee, through the WiFi interface.
- *Vision*, which provides the whole procedure of image analysis. The component first captures the image from the robots camera, the applies a color segmentation procedure to reduce the color-space to the few colors

Algorithm 5.1 Attacker Execution Algorithm

```
Stand Up
Calibrate Camera
loop
  if GameState = PLAYING then
    Process Image
    if BallFound then
      if BallDistance>0.25m then
        Move Towards Ball
      else
        if |BallXi - KickPosX| > 0.025m $\vee$ |BallY - KickPosY| > 0.025 then
          Fine Approach Ball
        else
          Kick Ball
        end if
      end if
    else
      Search for Ball
    end if
  end if
end loop
```

Algorithm 5.2 Goalkeeper Execution Algorithm

```
Stand Up
Calibrate Camera
loop
  if GameState = PLAYING then
    Process Image
    if BallFound$\wedge$BallDistance>0.8m then
      if $15^\circ$<|BallBearing|<$25^\circ$ then
        Fall to Ball Direction
      else
        Dive to Ball Direction
      end if
    else if BallFound then
      Track Ball
    else
      Search for Ball
    end if
  end if
end loop
```

that are used in a RoboCup scenario and, finally, applies object detection algorithms. At its current state, the component can detect and provide distance and bearing information only for the, but additional object detection is work in progress.

- *MotionController*, provide a convenient interface for implementations of motion patterns. Supports an omni-directional walking commands, Cartesian space walk as well as capable for realising complex motion skills (a.k.a special actions), in a reproducible way, such as kicks, dives and falls. Moreover, MotionController auto-detects if the robot has fallen down so a stand-up routine to be called.
- *Sensors*, a module that captures sensor data ordinated from the robot and publish them into other execution threads as well as the network, if needed. Its functionality is vital as it enable other components to access not only the current value of a sensor but a short history of values, which is very important for accurate computations — i.e. given an image, the head-roll angle is required to calculate the ball bearing correctly; if there is a time-skew between the image capture and the capture of the sensor value the calculation will not be accurate.
- *LedHandler*, is a code component for managing the robot's LEDs. While the module does not provide a fancy functionality, it simplifies the control API.

5.3 NaoQi Implementation

As a first part of our evaluation process, we implement the described agents (Section 5.2) directly on the NaoQi framework. NaoQi is provided from Aldebaran, the robot's manufacturer, as the appropriate framework for developing applications. Except from the API for interfering with the robot, it provides a modular architecture and capabilities from agent instantiation. We include it in the evaluation section to act as reference point and to highlight the problems that decrease the efficiency of development, and lead us on creation Monas.

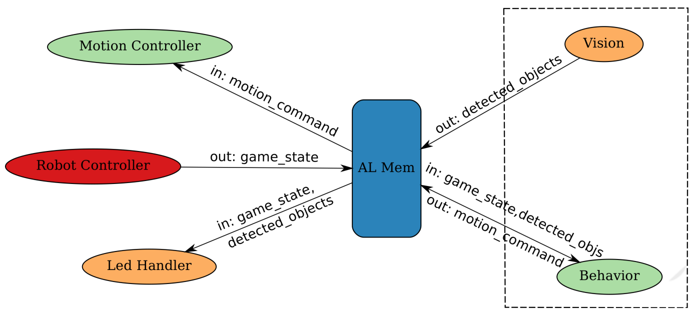


Figure 5.1: Implementation using the provided NaoQi framework.

For the implementing the two agents in NaoQi, we decided to create the four modules: the *Robot Controller, Led Handler, Motion Controller, and Vision & Behavior* module. It should be noted that the *Vision & Behavior* module is a combination of the *Vision* with a concrete section that implements the decision making for achieving the desired behaviour. The module implements two different roles, *Goalie* and *Attacker*, and the developer can choose which one to use at the robots initialization. The design is considered compact as each of the proposed modules can't be joint with an other module. That is because it may need to run on different frequency form the rest modules, i.e the *Motion Controller* checks if the robot has fallen of faster than the behaviour needs to execute for determining the next action, or it requires data from modules that run on different frequencies, as in the *Led Handler's* case, or executes blocking calls, as the *Robot Controller* does. We used NaoQi's *AL Memory* communication mechanism, for exchanging data between modules. The resulting system is depicted graphically in Figure 5.1.

5.4 Monas Agent Design

For evaluating the Monas agent instantiation architecture, we used the provided software components to create agents. Each of the components had to be modified and modelled as *"activities"*, so that can be instantiated within the architecture, as long as make use of the *Narukom* system for inter-object communication. Additionally, two new *Behavior* modules ware created, to implement the desired robot behavior, the *BehaviorGoalie* and *BehaviorPlayer*. Both modules, are responsible for the decision making process, as well as sending the appropriate commands to the *Motion Controller*. Finally, we decided to instantiate each activity as an agent, selecting appropriately between *BehaviorGoalie* and *BehaviorPlayer*. This approach enables us to fine-tune the execution frequencies of each activity, and thus, increase the system's efficiency.

5.5 ASEME Design Process

This section demonstrates the ASEME development process for our RoboCup team. We decided to skip the requirements analysis phase as the business modeling is quite simple; the team players have one collective goal, to win the game. In this section, except from the simple attacker player that is able to approach the ball and kick it, going one step further and taking advantage of the ASEME multi-agent co-ordination capabilities, we implement an updated attacker player, which coordinates with other players in order to implement an attack protocol. The analysis, design and implementation phases are described in detail below.

5.5.1 Analysis Phase

During the analysis phase in the societal level of abstraction, the developer needs to identify the roles that the robots may assume in the game and the protocols of interaction (AIP model) between these roles. Then, for each concrete role (i.e. a role that will be implemented as an agent), an individual System Roles Model is defined. Finally, the project team decides on the technologies that will be used for each identified functionality. For the RoboCup soccer team there are two concrete roles: the `player` (two players move freely around the field to attack and/or defend) and the `goalie` (the goalie stays near the team's goal posts and tries to prevent the other team from scoring).

Agent Interaction Protocols

During this analysis activity, the analyst defines what each role does within a given protocol. As both the player and the goalie are primitive and does not collaborate with each other, as described in Section 5.2, no protocols are needed to be defined. For the case of the advanced attacker player, in which we can unfold the advantages gained by integrating with ASEME, an attack protocol is designed and illustrated in Table 5.1. The two players participate by assuming the roles of `center` and `center_for` (one role each). The process followed by each role differs. When the `center` role is assigned to the agent, it supposes to go to the ball and if possible and pass it to the `center_for`. On the other hand, the `center_for` agent, move towards the opponent goal and when it take the control of the ball (if it has been passed correctly), shoots it to the goal (if there is a clear path). Gaia operators [WJK00] are used to create the liveness formulas that define the process of the role. The resulting formulas are presented at the last row of Table 5.1. Finalizing the protocol, requires to specifically define the engagement rules as well as the possibles outcomes. The protocol is engaged when no robot has control of the ball *and* `center` is the robot closest to the ball *and* `center_for` is the robot farthest from the ball. The possible outcomes can varying from the `center_for` roles shooting the ball to goal *or* an opponent taking control of the ball *or* the ball going out of bounds (e.g. due to an inaccurate pass) The rule for engaging in these roles is depicted in the second row of Table 5.1, followed by the expected outcomes in the third .

System Roles Models Definition

The system roles model defines each concrete role with a liveness model including the processes of the protocols in which it participates. To build the formula, we have to distinguish and recored the special needs of each role. Furthermore we have to organize the needs in groups and set which of them are needed to be executed concurrently, which continuously, which each in a

Table 5.1: The AIP model for the Attack protocol.

Participants	*center*	*center_for*
Engagement Rules	No robot has control of the ball *and* *center* is the robot closest to the ball *and* *center_for* is the robot farthest from the ball	
Outcomes	One of the *center_for* roles shoots to goal *or* an opponent takes control of the ball *or* the ball goes out of bounds	
Process	WalkTowardsBall. [passBall]	WalkTowardsGoal. [WalkTowardsBall. [kickBall]]

Role: goalie

Protocols: N/A

Liveness:

goalie = RobotController $^{\omega}$ || LedHandler $^{\omega}$ || MotionController $^{\omega}$ || (Stand . SendCalibration . active)

active = Sensors $^{\omega}$ || Vision $^{\omega}$ || decision $^{\omega}$

decision = WaitForBallMessage . (SearchBall | takeAction)

takeAction = TrackBall | action

action = RightFall | LeftFall | RightDive | LeftDive

Figure 5.2: The SRM model for the *goalie*.

specific order etc. covering the full range of Gaia operators. For the RoboCup players, we have find that each player, regardless of its role, have to concurrent and continuously execute the RobotController, LedHandler and MotionController modules, while the Sensors and Vision modules are needed to be executing only when the game state is set to *Playing* (a brief description of the functionality of each of the above modules is presented in Section 5.2). These source code modules are used directly in the resulting formulas, in conjunction with new ones, in order to achieve the agent behaviour described in Section 5.2. The formula for the goalie is presented in Figure 5.2, whereas Figure 5.3 for the attacker.

Role: attacker player

Protocols: N/A

Liveness:

player = RobotController $^{\omega}$ || LedHandler $^{\omega}$ || MotionController $^{\omega}$
|| (initialize . active)

initialize = Stand . SendCalibration

active = Sensors $^{\omega}$ || Vision $^{\omega}$ || decision $^{\omega}$

decision = WaitForBallMessage . (SearchBall | takeAction)

takeAction = TrackBall . (WalkTowardsBall | AlignWithBall | kickBall)

kickBall = LeftKick | RightKick

Figure 5.3: The SRM model for the `attacker player`.

The extended liveness formula for the advanced attacker, with co-ordination capabilities, is shown if Figure 5.4. Note that in this scenario, the attacker can participate in the Attack Protocol (see Section 5.5.1) either as a `center` or as a `center_for`. It should be mentioned that the SRM model is exactly the same as in the basic attacker role (Figure 5.3) up to the point of integration to the Attack Protocol. The protocol is inserted smoothly into the model by appending it, without the need of extensive modification. The agent's conformity to the Attack Protocol is ensured as the implementation of the protocol, is neither re-designed nor reinvented, but instead is provided by the AIP model itself. This novel feature of ASEME is one of the reasons that lead in its selection by this thesis.

The Functionality Table

In the SRM each activity that participates in the liveness formula is also associated with a functionality, therefore the next step is to build the functionality table (FT). This table helps the development team to identify the competencies needed and to decide on the technology to be used. In Figure 5.5 the reader can see the FT for the `goalie` role whereas in Figure 5.6 represents both the `attacker player` and the `extended attacker player` implementations.

Role: extended attacker player

Protocols: Attack: center, Attack: center_for

Liveness:

player = RobotController $^{\omega}$ || LedHandler $^{\omega}$ || MotionController $^{\omega}$ || (initialize . active)

initialize = Stand. SendCalibration

active = Sensors $^{\omega}$ || Vision $^{\omega}$ || decision $^{\omega}$

decision = WaitForBallMessage. (SearchBall | takeAction)

takeAction = TrackBall . (WalkTowardsBall | AlignWithBall | kickBall | center | center_for)

kickBall = LeftKick | RightKick

center = WalkTowardsBall. [passBall]

center_for = WalkTowardsGoal. [WalkTowardsBall. [kickBall]]

Figure 5.4: The SRM model for the `extended attacker player`.

We decide to present both attachers in the same figure as it is clear that except from the `center` and `center_for` sections, the rest of the figure remains exactly the same.

Although in our approach there are many activities, there are only three basic functionalities. The first one, is to interface directly with the Nao robot libraries, which provide access to the whole supported API. This method, provides the fastest way to interact with the robot but is subject to both version updates from Aldebaran as well as the lost of platform independence. The second functionality that is present on the FT, is the Monas's Hardware Abstraction Layer which supports a constant interface for building activities while it can be used with different underlaying platforms. The functionality, although it is linked to the Nao Libraries, is done with a transparent to the user way. Narukom, the third available functionality, is the publish/subscribe blackboard communication system, which has been the successful candidate for Monas's communication system. It is responsible for circulating the information among different activities, robots and computers in general.

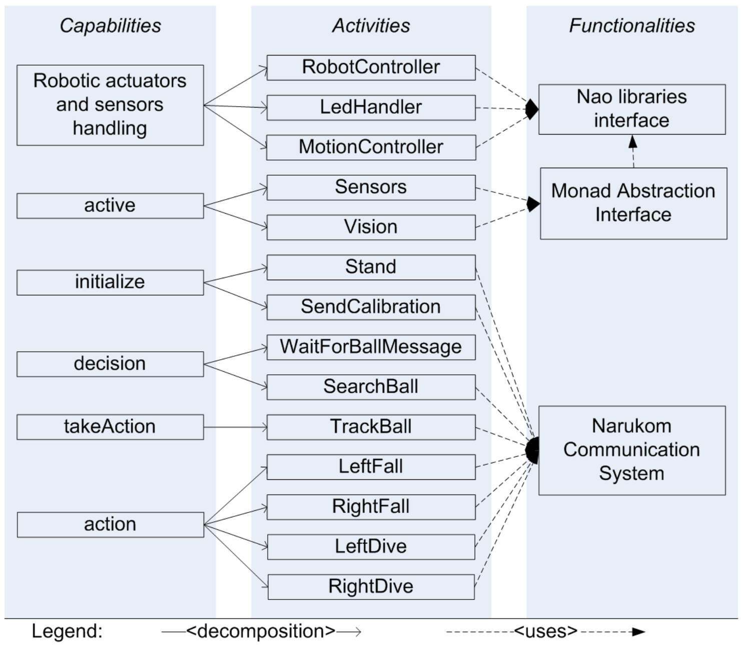


Figure 5.5: The functionality table for the `goalie` role.

It should be noted, that in both cases (goalie and player), there has been an important abstraction in the design from the robotic hardware. Developers are interact with the hardware only for low-level tasks, as opposed to the rest of the design which interact with the communication system. That provides the required functionality for writing more coherent source code, which is easier to maintained and developed.

5.5.2 Design Phase

In the design phase, the ASEME SRM2IAC tool can be used to transform the SRM model to an EAC or an IAC model. The EAC and IAC models are statecharts where the developer can insert events, conditions, and actions in the transition expressions, thus controlling each role's process either in a protocol (in the EAC model) or for coordinating its capabilities (in the IAC model). Hence at this development phase, the behaviour of the system is finalized. Implementation phase, which is the remaining phase for realizing the system, is

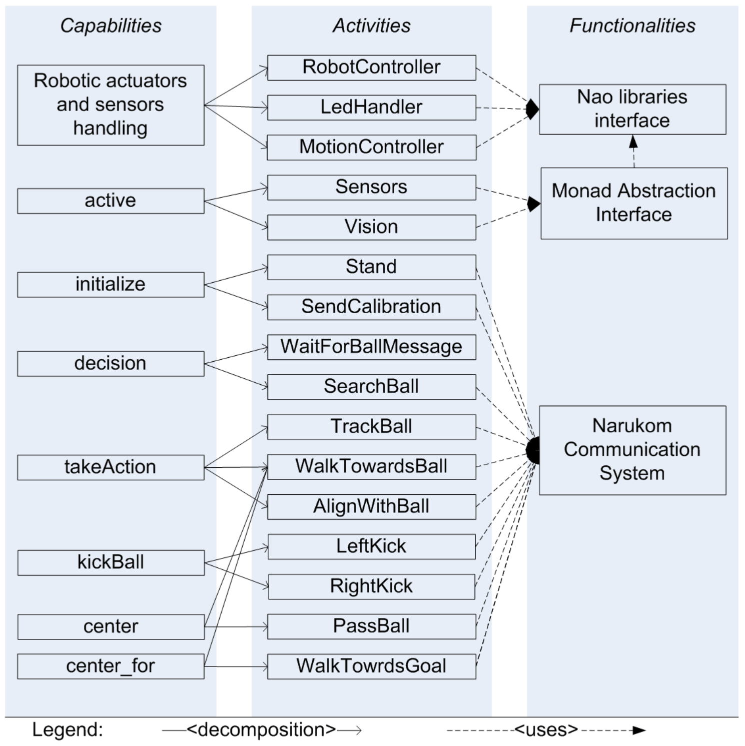


Figure 5.6: The functionality table for both the `attacker player` and the `extended attacker player`. The sections `center` and `center_for`, apply only on the later.

an one-to-one transformation in which modifications of the design models are restricted.

Inter-Agent Control Model

The SRM2IAC tool can be used to transform the process part of the agent interaction protocol model to a statechart, namely the inter-agent control model (EAC). A state diagram is generated by an initial AND-state named after the protocol. Then, all participating roles define OR sub-states. The right hand side of the liveness formula of each role is transformed to several states within each OR-state by interpreting the Gaia operators [SM09].

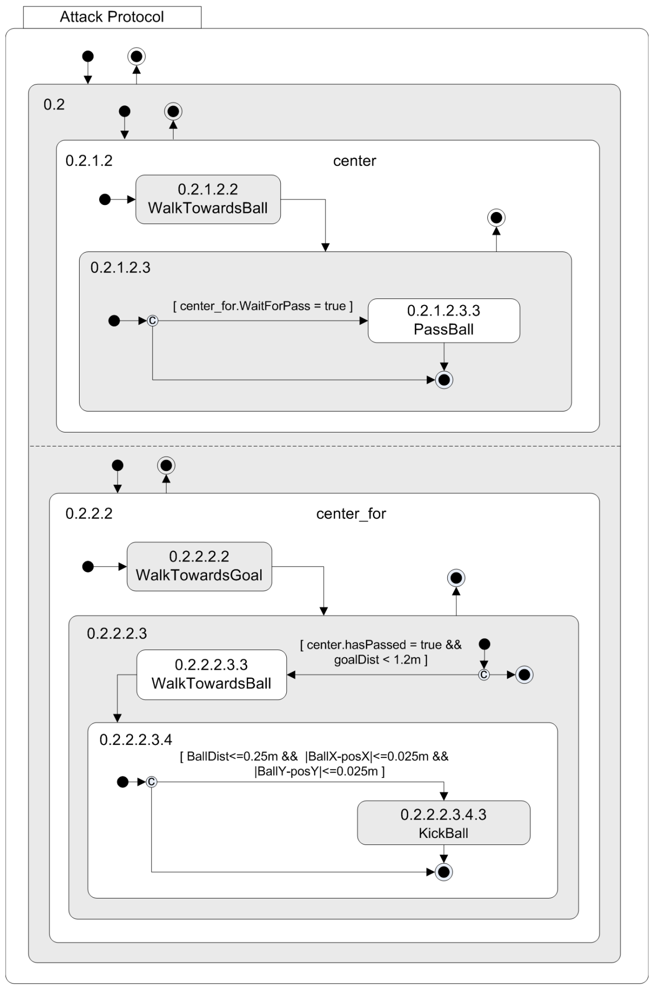


Figure 5.7: The EAC model in a graphical representation using the schematics of statecharts.

Currently, the only developed protocol is the `Attack Protocol`, which concerns the only the `extended player`, thus we must transform the expression "attack_protocol = center || center_for" followed by the processes of the two participating roles. The result is depicted graphically in the form of a statechart (see Figure 5.7). The modeler had to define transition expressions for all the transitions in order to realize the appropriate functionality and the possible outcomes, as they were defined in `AIP` model of the `Attack Protocol` (Figure 5.4).

Intra-Agent Control Model

The final step in the design process, is to generate the IAC model for each agent. The model can capture multiple roles, as they have been defined in the previous step of the SRM model creation. The intra-agent control model is also initiated by the SRM2IAC tool for each role. The resulting model is not a finished design, but has to be completed with transition expressions, on each transition, manually by the modeler. Transition expressions must follow the syntax defined in Table 4.6. This step requires the most attention by the modeler, as wrongly defined expressions can lead to incorrect or unexpected behaviour on the run-time execution of the statechart. Finally, the modeler has to fine-tune the expressions to achieve better efficiency from the system.

Designing the transition expressions for both players, faces some common challenges. First of all, an important separation should be made between the structural and the behavioural components of the system, as IAC, model them in the same representation. While this introduction is a novelty of IAC, is still must be considered as a separate task for the modeler because the selection of the appropriate expressions have an great impact on the system's efficiency and performance. The system components are common on both player are the, already developed components described in Section 5.2, namely the *RobotController*, *LedHandler*, *MotionController*, *Sensors* and *Vision*. Additionally, apart from these components, the *decision* component can also be considered as structural, as it's functionality as a whole, describes the behavior of the system. The transition expressions in these components, are mainly timeout conditions — paired with start-timeout actions — to manage the system's reactiveness and performance, as long as CPU time distribution. In addition, can control the execution of a component when a condition is active, i.e. when the games state equals *Playing*.

For the behavioural components, the modeler is based on the agent's execution algorithms, as being defined in Section 5.2, in order to implement the same behaviour. The conditions that change the agent's behaviour, such as *WalkTowardsBall* when the ball distance is less than 0.25 meters, can be

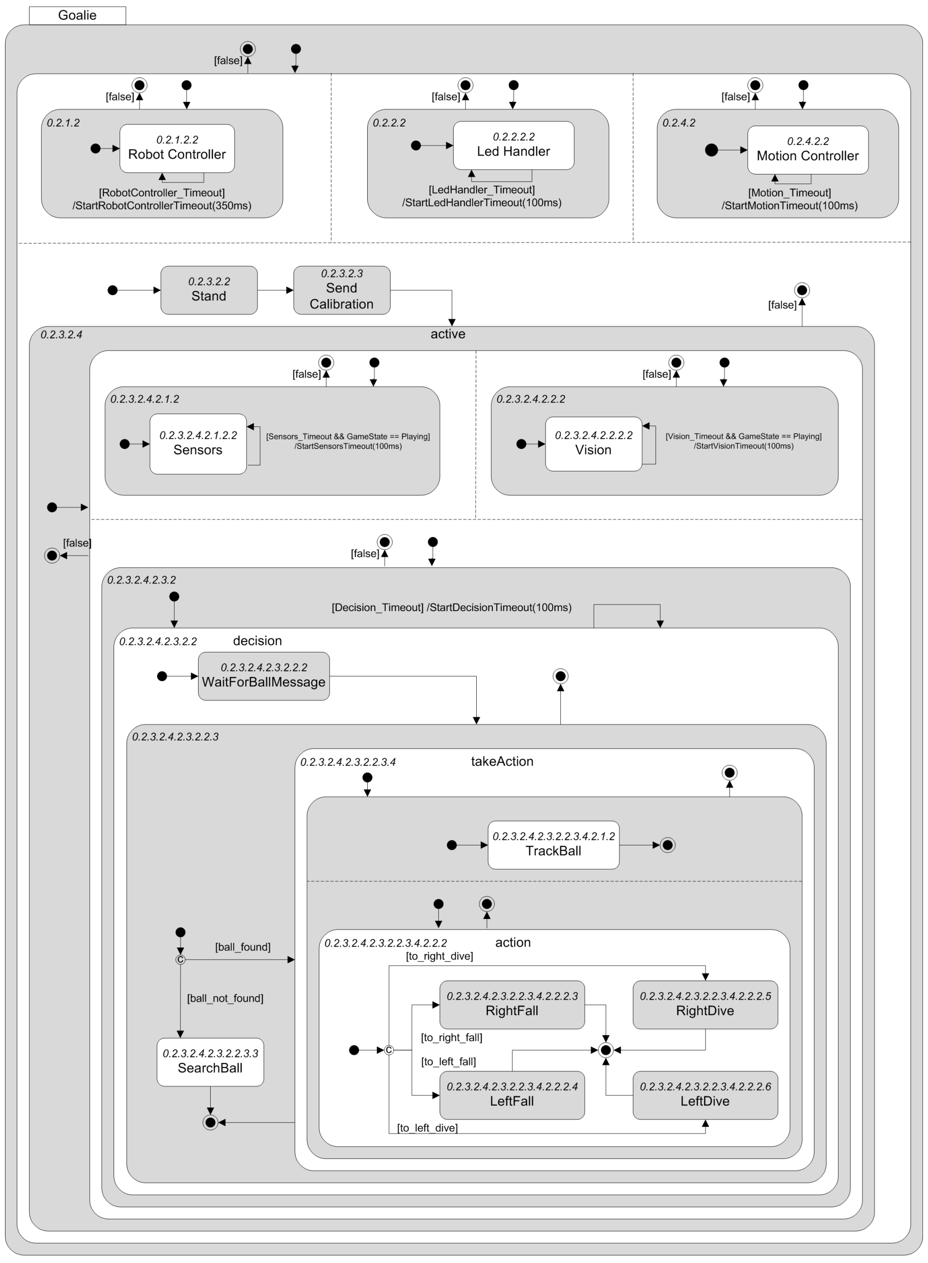


Figure 5.8: The IAC Model (statechart) for the `goalie`.

taken directly from the algorithms. As a result, Figure 5.8 and Figure 5.9 shows IAC model for the `goalie` and `player` respectively.

In the case of the `extended player`, the modeler has to integrate the `Attack Protocol` with the already created design of the `player` (Figure 5.9).

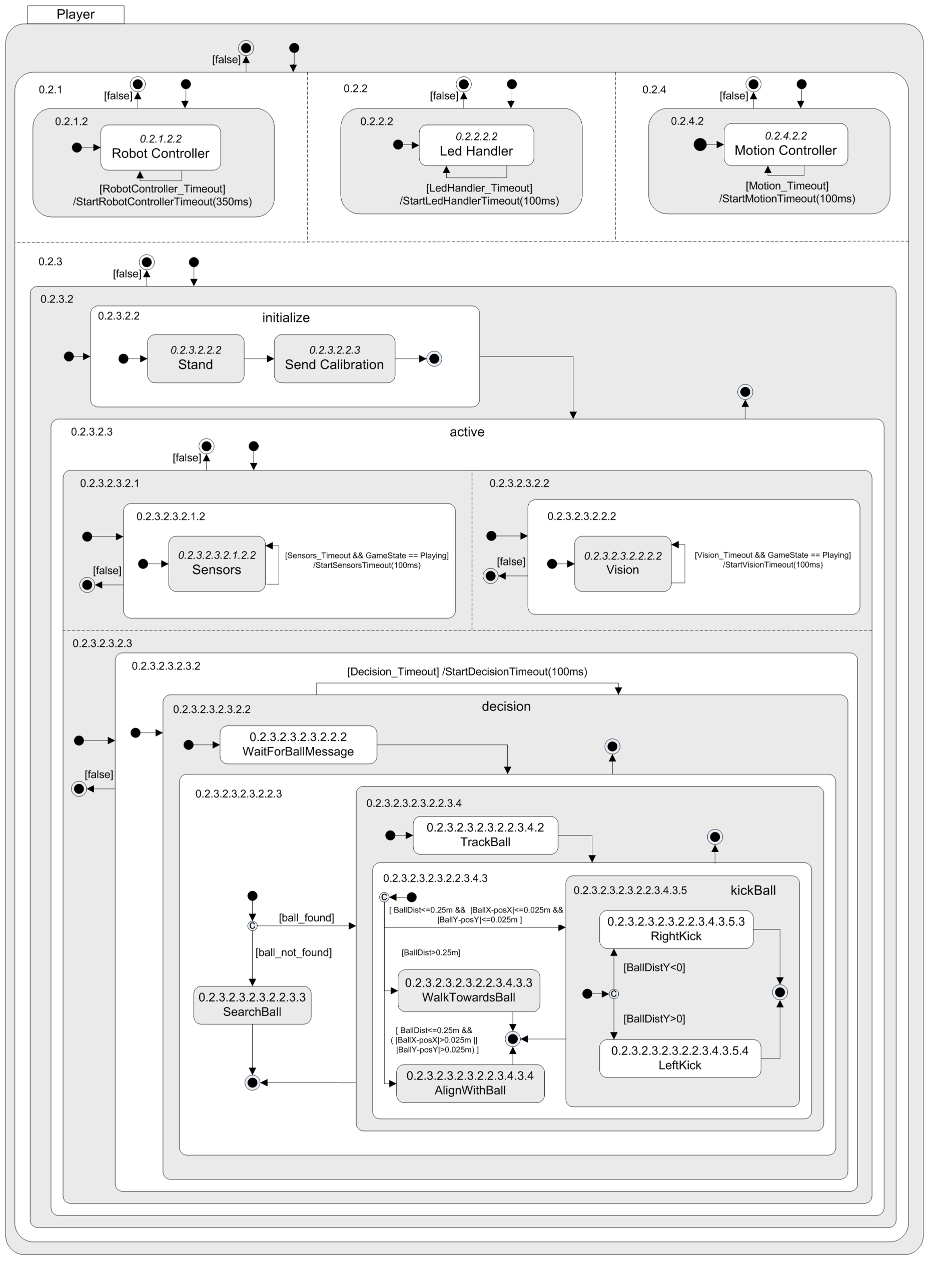


Figure 5.9: The IAC Model (statechart) for the `player`.

As both IAC and EAC models are based on statecharts, it can be done by directly inserting the EAC into the IAC, at the correct position, with little to no modifications. The method provides a significant advantage: the agent is ensured that will implement the co-ordination protocol correctly. To finalize the design, only the engagement rules of the protocol must be implemented as conditions on the agent's statechart (IAC), in order to activate the protocol. The design is illustrated in Figure 5.10.

5.5.3 Implementation Phase

As a last step in ASEME process, comes the implementation phase, responsible for the realization of the agent. At this point, the designed IAC models which describe the agents are transformed, using the IAC2Monas process (which analyzed in Section 4.3.2). The process provides the appropriate mechanism to automatically transform the IAC model into source code, compatible with, the Monas's architecture, statechart engine. The platform-independent IAC model must be transformed to a platform-dependent one (adhering to the Nomad architecture) and to executable code. After the transformation and building the generated source code, the has been fully realized and is ready to run on the selected hardware (physical) platform.

5.5.4 Methodology Comparison

Comparing the implemented methodologies, using the metrics defined in Section 5.1, we can see the pros ans cons that it methodology offers. First of all, using NaoQi directly, without *Monas*, we observe a performance decrease. That was because of the overload of *AL Memory* system in the exchanging of information between the modules. Developing times are higher in the *Monas*, as is *NaoQi* because of the more complex communication interface. While *ASEME* should score less, due to the lack of graphical tools, it requires more time than the other two methodologies. In parenthesis, is the total time minus

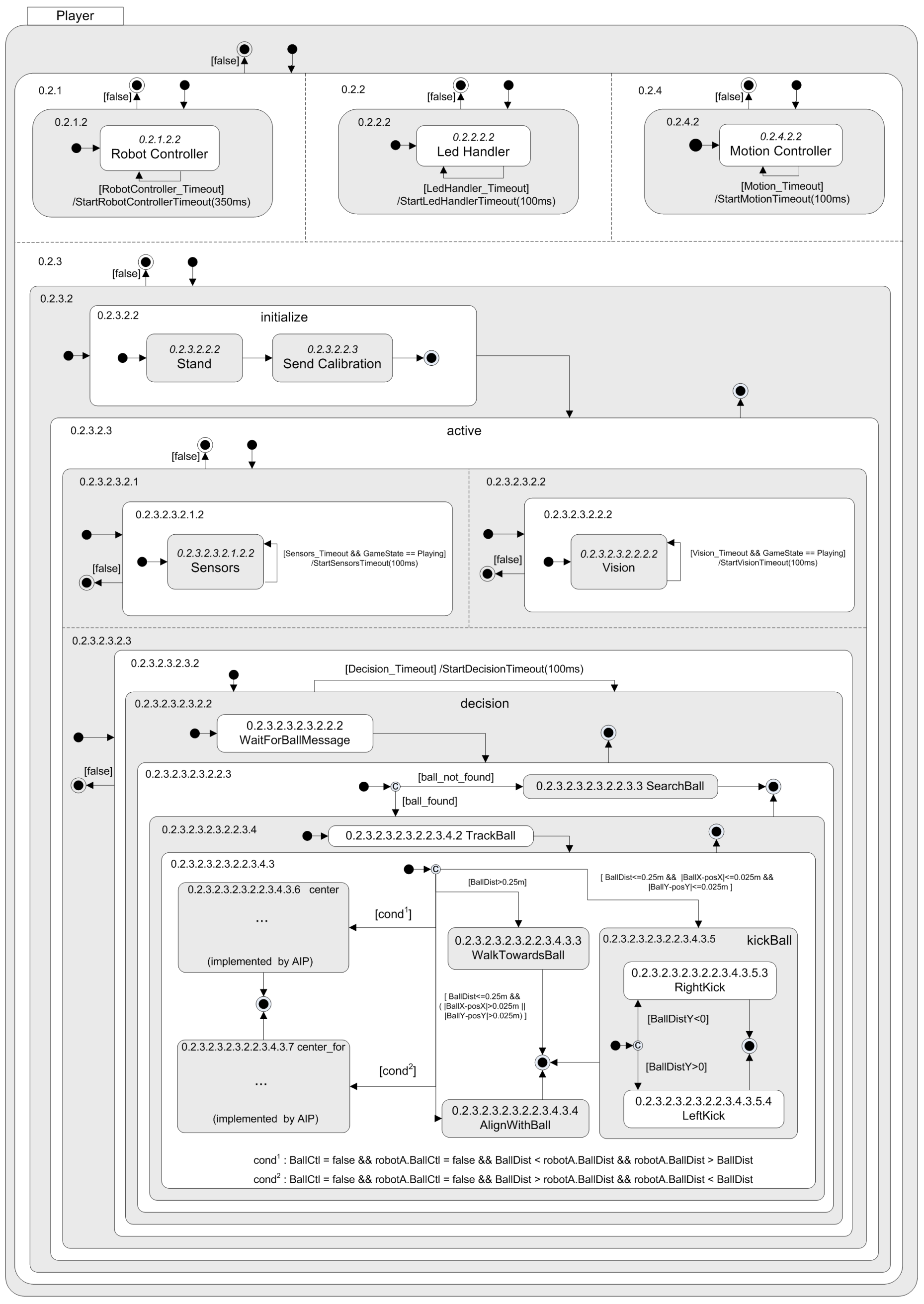


Figure 5.10: The IAC Model (statechart) for the `extended player`.

Table 5.2: Comparison of evaluated methodologies

Metric/Concept	NaoQi	Monas Agents	ASEME Agents
Run-Time Performance	low	high	high
Developing Process			
Total Developing Time	7	9	12(5)
Source Lines of Code	390	486	826
of them auto-generated	N/A	N/A	760
Num. of State Variables	18	18	N/A
Code Coupling	high	low	low
Code Cohesion	low	low	high
Debugging Process			
Debugging Time	13	15	5
Num. of Bugs	16	16	4
Type of most common Bugs	logical	logical	syntactical

from the time required to fill the *IAC* model with transition expressions. On the lines of code, *ASEME* methodology was require, by far, the fewest lines, as it is based on code auto-generation. *Monas* requires slightly more lines, mainly because of *Narukom*. The implementations of behavior modules on both *NaoQi* and *Monas*, require the developer to control a great number of global variables that define the current state of the agent, as opposed to *ASEME* that such a concept is not applicable. Developing on the *NaoQi* platform directly, leads to a high degree of code coupling as even a small change on the API, especially the *AL Memory* interface, will propagate a series of modifications on the source code. On code cohesion, both *NaoQi* and *Monas* score low, as the behavior modules are build as monolithic components, and the source code is not distributed. *ASEME* instead uses statechart formalism to model both the system and the behavior, and divides the source code in activities and functionalities. The above, are reflected on the debugging process, in which *NaoQi* and *Monas Agents* have increased debugging times, with most of the bugs found at the behavior code segments. As opposed to *ASEME*, most of the bugs were logical, and thus harder to detect.

Chapter 6

Conclusion

Conclusion: end or termination, the close. But that would simply be unfair for *Monas*, who has just given birth. In this thesis, we try to cover many aspects relative to robotic development but the task is far from reaching the finishing point. *Monas* can, and should, be further developed and, hopefully, have a long journey to go.

6.1 Future Work

In this thesis, we designed *Monas* to be a complete architectural framework and to address as many issues as possible related to robotic development. But as the task is far from over, further work is required to be done, in order to make the architecture more flexible, optimized and expand it, as well as to attract third-party developers to work with it from the open source community.

First of all, while *Monas* is designed to be robot independent, it is only configured for use with the *Nao* robotic platform. Additional platforms can be added to the source tree, so that developers and researchers, which are currently working on other platforms, can be attracted and evaluate the architecture.

Additionally, while testing the *Monas* on the *RoboCup* competition, we find out that the team wasn't using the provided interfaces that abstract from the robotic hardware, and thus allow a seaming-less robot replacement, but was using the API provided from the robot's manufacture directly. Consequently, the provided interfaces failed to capture the special requirements that were needed and design analysis must be done to improve them. A new approach on the *robot configuration* part, can also be completed with a kinematic library, as it will boost the developing process.

Finally, the statechart engine can be improved. *Condition connectors* supported by *UML* can be implemented, promoting the engine as a standalone application that is not bounded to *IAC* metamodel. Also, as far as the system modelling concerns, the engine can support a priority scheme, so that activities with higher priority scheduled for immediate execution instead of waiting in the queue. One last addition, could also be a non-deterministic execution of transitions, with or without a priory probability, when a transitions conflict occurs.

6.2 Lessons Learned

During the progress of this thesis, the first lesson that i learned, and that was done "the hard way", was not to underestimate the so-called *"system work"*. When the developing process takes places on a remote system, with different operating system, which needs a cross-compiler and low-level developing has to be done the human work-hours are increasing exponentially.

Furthermore, the second lesson that i thought while developing the thesis, was that *there is no such think as a generic, optimized system!* Instead, a there is a trade-off between generality and optimization: the more optimize the system is for the target platform — and must be optimized a lot while developing for robotic systems — the greater the loss on generality.

Finally, this thesis provide me experience on team-work, as it was used in *RoboCup* team *Kouretes*. This experience is proven to be very valuable as it is clearly stated by *Andrew Carnegie*: “Teamwork is the ability to work together toward a common vision. The ability to direct individual accomplishments toward organizational objectives. It is the fuel that allows common people to attain uncommon results”.

Bibliography

[Ale01] Andrei Alexandrescu. *Modern C++ design: generic programming and design patterns applied*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 2001.

[BBM03] Frank Budinsky, Stephen A. Brodsky, and Ed Merks. *Eclipse Modeling Framework*. Pearson Education, 2003.

[BRJ99] G. Booch, J. Rumbaugh, and I. Jacobson. The Unified Modeling Language User Guide. *Reading, PA: Addison-Wesley*, 1999.

[CD07] M.L. Crane and J. Dingel. UML vs. classical vs. rhapsody statecharts: not all models are created equal. *Software and Systems Modeling*, 6(4):415–435, 2007.

[ČN04] K. Čapek and C. Novack. *RUR (Rossum's universal robots)*. Penguin Group USA, 2004.

[FG97] Stan Franklin and Art Graesser. Is it an agent, or just a program?: A taxonomy for autonomous agents. In Jörg Müller, Michael Wooldridge, and Nicholas Jennings, editors, *Intelligent Agents III Agent Theories, Architectures, and Languages*, volume 1193 of *Lecture Notes in Computer Science*, pages 21–35. Springer Berlin / Heidelberg, 1997. 10.1007/BFb0013570.

[Fou10] The Eclipse Foundation. *Xpand Reference Manual*, 2010. `http://wiki.eclipse.org/Xpand/`.

[GB06] David Gouaillier and Pierre Blazevic. A mechatronic platform, the Aldebaran robotics humanoid robot. *32nd IEEE Annual Conference on Industrial Electronics, IECON 2006*, pages 4049–4053, November 2006.

[GFC09] José Manuel Gascueña and Antonio Fernández-Caballero. Towards an integrative methodology for developing multi-agent systems. In *ICAART*, pages 392–399, 2009.

[GHJV95] Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides. *Design patterns: elements of reusable object-oriented software*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 1995.

[GP00] C.F. Goldfarb and P. Prescod. *XML handbook*. Prentice Hall PTR Upper Saddle River, NJ, USA, 2000.

[Gro05] Object Managment Group. *OMG Unified Modeling Language Specification*, 2005. http://www.omg.org/spec/UML/2.0/.

[Har87] David Harel. Statecharts: a visual formalism for complex systems. *Science of Computer Programming*, 8(3):231 – 274, 1987.

[Hem05] P. Hemenway. *Divine proportion: Phi in art, nature, and science*. Sterling, 2005.

[HK04] David Harel and Hillel Kugler. The rhapsody semantics of statecharts (or, on the executable core of the uml) - preliminary version. In *SoftSpez Final Report*, pages 325–354, 2004.

[HN96] David Harel and Amnon Naamad. The statemate semantics of statecharts. *ACM Trans. Softw. Eng. Methodol.*, 5(4):293–333, 1996.

[KAK$^+$97] Hiroaki Kitano, Minoru Asada, Yasuo Kuniyoshi, Itsuki Noda, Eiichi Osawa, and Hitoshi Matsubara. Robocup: A challenge problem for AI. *AI Magazine*, 18:73–85, 1997.

[KWB03] Anneke G. Kleppe, Jos Warmer, and Wim Bast. *MDA Explained: The Model Driven Architecture: Practice and Promise*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 2003.

[Liu04] C.G. Liungman. *Symbols–Encyclopedia of Western Signs and Ideograms*. Hme Pub, 2004.

[MH08] Ken Martin and Bill Hoffman. *Mastering CMake 4th Edition*. Kitware, Inc., USA, 2008.

[Mur03] Jan Murray. Specifying agent behaviors with uml statecharts and statedit. In *RoboCup*, pages 145–156, 2003.

[PBBY04] Daniel Polani, Brett Browning, Andrea Bonarini, and Kazuo Yoshida, editors. *RoboCup 2003: Robot Soccer World Cup VII*, volume 3020 of *Lecture Notes in Computer Science*. Springer, 2004.

[RN03] Stuart Russell and Peter Norvig. *Artificial Intelligence: A Modern Approach*. Prentice-Hall, Englewood Cliffs, NJ, 2nd edition edition, 2003.

[Rob10] RoboCup Technical Committee. Standard Platform League rule book, 2010.

[SA04] Herb Sutter and Andrei Alexandrescu. *C++ Coding Standards: 101 Rules, Guidelines, and Best Practices (C++ in Depth Series)*. Addison-Wesley Professional, 2004.

[SM08] Nikolaos I. Spanoudakis and Pavlos Moraitis. The agent modeling language (amola). In *AIMSA*, pages 32–44, 2008.

[SM09] Nikolaos I. Spanoudakis and Pavlos Moraitis. Gaia agents implementation through models transformation. In *PRIMA*, pages 127–142, 2009.

[SM10] N. Spanoudakis and P. Moraitis. Model-driven agents development with aseme. In *Proceedings of the 11th International Workshop on Agent-Oriented Software Engineering (AOSE 2010)*, pages 49–60, 2010.

[Spa09] Nikolaos Spandoudakis. *The Agent Systems Engineering Methodology (ASEME)*. PhD thesis, Universite Paris Descartes, 2009.

[Str00] Bjarne Stroustrup. *The C++ Programming Language*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 2000.

[Sut04] Herb Sutter. *Exceptional C++ Style: 40 New Engineering Puzzles, Programming Problems, and Solutions*. Pearson Higher Education, 2004.

[Tho10] Lee Thomason. *TinyXML Reference Manual*, 2010. `http://www.grinninglizard.com/tinyxmldocs/index.html`.

[Vaz10] Evangelos Vazaios. Narukom: A distributed, cross-platform, transparent communication framework for robotic teams. Diploma thesis, Technical University of Crete, Greece. Department of Electronic and Computer Engineering, 2010.

[(W308] The World Wide Web Consortium (W3C). *Extensible Markup Language (XML) 1.0 (Fifth Edition)*, 2008. `http://www.w3.org/TR/REC-xml/`.

[WJK00] Michael Wooldridge, Nicholas R. Jennings, and David Kinny. The gaia methodology for agent-oriented analysis and design. *Autonomous Agents and Multi-Agent Systems*, 3(3):285–312, 2000.

[WP04] Michael Winikoff and Lin Padgham. *Developing Intelligent Agent Systems: A Practical Guide*. Halsted Press, New York, NY, USA, 2004.

[Yag03] Karim Yaghmour. *Building Embedded Linux Systems*. O'Reilly Media, Inc., 2003.