An Intuitive and Flexible Architecture for Intelligent Mobile Robots

A thesis presented

by

Xiao-Wen Terry Liu

to

The Department of Computer Science

in partial fulfillment of the requirements

for the degree of

Master of Science

in the subject of

Computer Science

The University of Manitoba

Winnipeg, Manitoba

October 2005

Thesis advisor

Dr. Jacky Baltes

Author

Xiao-Wen Terry Liu

An Intuitive and Flexible Architecture for Intelligent Mobile Robots

Abstract

The goal of this thesis is to develop an intuitive, adaptive, and flexible architecture for controlling intelligent mobile robots. This architecture is a hybrid architecture that combines deliberative planning, reactive control, finite state automata, behaviour trees and uses competition for behaviour selection. This behaviour selection is based on a task manager, which selects behaviours based on approximations of their applicability to the current situation and the expected reward value for performing that behaviour. One important feature of this architecture is that it makes important behavioural information explicit using *Extensible Markup Language* (XML) [99]. This explicit representation is an important part in making the architecture easy to debug and extend. The utility, intuitiveness and flexibility of this architecture is shown in an evaluation of this architecture against older control programs that lack such explicit behavioural representation. This evaluation was carried out by developing behaviours for several common robotic tasks and demonstrating common problems that arose during the course of this development.

Contents

List of Figures

Acknowledgments

I want to thank Dr. Jacky Baltes and Dr. John Anderson for their advice, support, and especially their patience these past few years. Their sage advice was indispensable and guided me through the darkest of times when writing this thesis. I also want to thank my family for their support, patience, and help pushing me when I needed it. I also want to thank the members of our little lab community of the Autonomous Agents Lab at the University of Manitoba with setting up the robots and environment when I needed help. Finally, I would like to thank all my employers and clients for financially supporting and providing me with the flexible work schedule I needed to finish my program. Without all your support, none of this would be possible. Thank you.

Chapter 1

Introduction

The goal of this thesis is to design an architecture for intelligent mobile robotics that is easier and more intuitive to modify, adapt, and extend than previous architectures. An *architecture* is a unifying, coherent form or method of construction, which provides the foundation for creating powerful intelligent systems. Due to the complexity of formally evaluating this architecture, which would require a formal user study, this thesis will use anecdotal evidence in the form of several common tasks to show the features of the architecture (see Chapter 4 for details). Similar to other intelligent systems, mobile robots must select correct actions out of a huge set of possibilities. As an example, a mail delivery robot needs to plan an efficient route to all mail drops (this problem is equivalent to the travelling salesman problem which is NP-hard) [25, 57, 95, 98]. To make matters worse, mobile robots need to act under constraints imposed by the real world. For example, a robot may have to react fast enough to avoid certain obstacles. Furthermore, sensors and actuators are noisy

and inaccurate (e.g. grainy pictures that leads to uncertainty in the identification of objects).

When discussing intelligent mobile robot, the meaning of *mobile* is clear: the robot must be able to relocate itself. However, the meaning of *intelligent* is less clear. The definition of intelligence has been debated and discussed by many researchers [8, 84, 93]. For the purposes of this thesis, an intelligent mobile robot must exhibit the following qualities:

- the mobile robot must act autonomously,
- the robot must perform appropriate actions in controlled and uncertain situations.

According to Russell and Norvig [84], acting autonomously means the robot will move and behave independently, free from direct human control. Performing the *appropriate* action means demonstrating behaviours that are working to the completion of the system's objective [84]. This thesis may often refer to the intelligence of the robot simply as "the robot".

Perception, reasoning, and execution (see Figure 1.1) are common to all intelligent mobile robots. Sensors such as light, sonar, or touch sensors gather information about the environment such as the robots approximate location, the location of nearby obstacles or other robots, and information about itself such as battery power levels. Raw sensor data must often be filtered, correlated and/or interpreted to form perceptions. For example, a group of pixels are smoothed and interpolated to generate the position of a ball or another object relative to the robot. The set of all perceptions guides the formation of the world model in which the robot reasons. Reasoning often involves

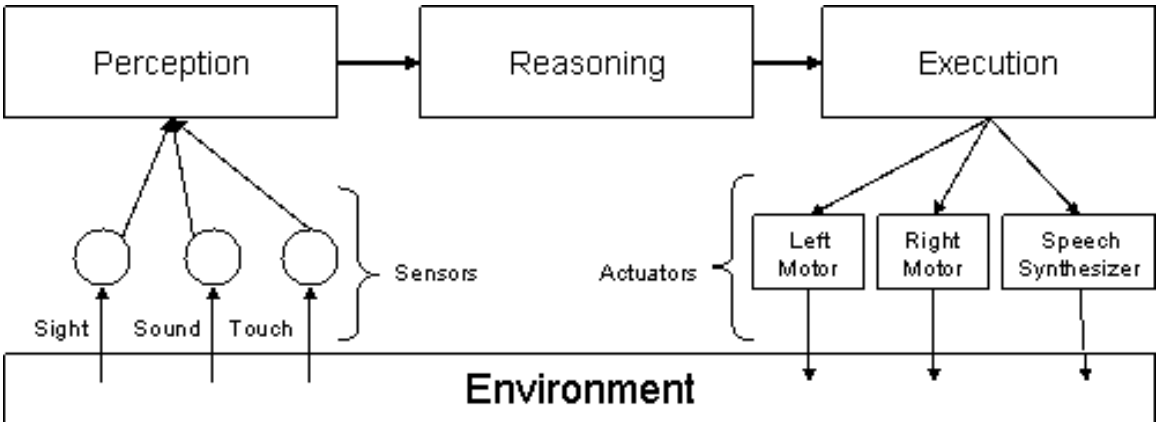


Figure 1.1: Generic Architecture.

the robot trying to generate and satisfy multiple and possibly competing goals. Commonly, the output of the reasoning stage is an abstract plan (e.g. a sequence of actions or behaviours) to achieve the robot's goals. In the execution stage, the abstract plan is implemented. Abstract operators (e.g. *drive-through-the-door*) are converted into lower level commands for motors and other actuators. One of the most interesting and most frustrating aspects of working with mobile robots is that there can be complete failure, or a range of other types of errors, in any of these stages. For example, in the perception stage sensors might fail, or multiple sensors may return conflicting data about the environment. In the processing stage, unexpected situations may cause the robot to decide to perform wrong behaviours or behaviours that are detrimental to it. In the execution stage, the actuator may fail or wheels may frequently slip (in the case of wheeled robots) on certain surfaces such as dirt, sand, or carpet. Developing, maintaining, and modifying systems to control intelligent mobile robots in the real world can be a daunting task.

The problem with most systems is that they are often limited by the initial design of the original developer or specifications. These systems can not cope with error and are not flexible enough to change significant aspects of the program. For example, transforming a soccer playing robot into a garbage collecting robot may require going through the control program and modifying perceptions (e.g. targets and obstacles), reasoning (e.g. approach a trash can from any direction vs. approaching a ball in only certain directions), and execution (e.g. add an additional actuator). The problem is that much of the necessary information is only available implicitly in the implementation of the system. Without making the necessary information explicit, the architecture and its implementation does not easily allow for reusability, and extensibility. A system with an architecture that utilizes an agent-based approach and makes relevant information explicit should be intuitive, flexible and extensible. One requirement of the architecture is for necessary information to be made explicit. However, the overall goals of this architecture is that it must be intuitive, flexible, adaptable, and extensible. The intuitiveness refers to the simplicity of the architecture for developers to understand and work with: specifically, how easy it is to add, change and remove behaviours. The flexibility refers to the architecture's ability to work in different domains and tasks. The adaptability refers to the architecture's support for different sensors/actuators and coping abilities to unexpected scenarios. The extensibility of this architecture (i.e. the ability to add/remove/modify behaviours) can be considered another requirement. However, extensibility can also be considered in terms of the architecture's intuitiveness to work with and its flexibility for different tasks.

The main contributions of this thesis are:

- a survey of existing robotic architectures;
- the design of a new intuitive and flexible architecture;
- a prototype implementation of the proposed architecture and;
- an evaluation study of the proposed architecture (via the implementation).

The prototype program required hundreds of lines of XML (*eXtensible Markup Language* [99]) specification as well as thousands of lines of C++ code. The survey of existing architectures, the design and implementation of the new architecture, and the evaluation study took several years (2003-2005) to complete. During this time, observations and participation at several international robotic competitions helped further motivate this research.

Although the need for an intuitive architecture is inherent to all intelligent mobile robot applications, examples from the domain of robotic soccer will be used since this domain shares many of the same qualities found in other domains (such as dynamic moving objects), and it has a strong and active community of researchers. The flexibility of this architecture will be demonstrated by using examples from other domains such as collection and racing robots. In section 1.1, architectural issues, which may seem abstract and vague, are made concrete through examples gathered during UM RoboCup teams' participation at past robotic soccer competitions. Chapter 2 describes previous work related to this research. Chapter 3 shows the design of the new proposed architecture with an explicit behaviour representation prototype, which

uses XML. Chapter 4 describes some of the implementation details and how the new proposed architecture is evaluated. Finally, Chapter 5 will conclude this thesis.

1.1 Motivation

Robotic soccer is an excellent testbed for research into intelligent mobile robotics. Playing soccer well requires solutions to many problems that are currently being actively investigated by researchers. For example, a player has to have a core set of skills: real-time control (e.g. the ability to kick the ball and accept passes), perception (e.g. the ability to see the ball), awareness (e.g. localization), strategy, coordination and communication (e.g. the ability to set-up plays). The soccer domain is interesting because of its complexity and dynamic nature. There are many factors to consider in making a successful soccer team that can play well and win games. Some factors such as making players stronger and faster are important, but making them smarter is also important. The intelligence of a player encompasses the general knowledge of the game, the plays or strategies to use during play, and the ability to adapt to situations whether they were previously encountered or are entirely new. Another important advantage of robotic soccer is that there is a large and growing community of researchers that participate at international competitions, such as the RoboCup [42] and the Federation of International Robosoccer Association (FIRA) [43] competitions. These competitive environments allows for quantitative evaluation of different approaches.

In the last three years (2003-2005), the University of Manitoba has fielded robotic soccer teams (e.g. [10, 11, 12]) at these events. Observations of the performance

of teams at these events has made the need for powerful architectures for mobile robots blatantly apparent. To be successful in the competition, the robotic controller used by a team must be adapted to different environments and different opponents. For example, a goalkeeper has a large set of possible behaviours, which are selected based on the opponent's style of play. For example, for opponents with kickers (some teams can kick a ball at up to nine meters per second), behaviours that are fast-acting and project the position of the ball based on its current velocity are necessary. However, these fast-acting behaviours are very susceptible to noise and therefore if the opponents do not have kickers, behaviours that are slow-acting and based on repeated measurements of the ball's position are more appropriate. Thus, the influence of the fast-acting behaviours needs to be increased/reduced to make the robot's behaviour more predictable and reliable when playing against teams with/without kickers.

For the UM 2003 RoboCup control program [10], team members needed to sift through many lines of C++ code to figure out exactly in what contexts to modify a behaviour. Extra time was needed to test the effect of this change on the whole system because of the complex interaction between multiple behaviours. Even in the C++ code, it was difficult to determine what other behaviours depended upon a given behaviour. This indicates a clear need for the architecture to be flexible in these situations.

Another common problem was that behaviours had to be modified to compensate for noise in the sensors. For example, the minimum distance between two robots (e.g. when building a "defensive wall" to block a shot) depended on the quality of the location information. If the location data is very accurate the distance between

robots can be reduced (thus reducing the chance of the opponent team scoring), but if the location information was too noisy, robots would collide with each other resulting in no defensive wall at all.

In another example, the robot may unintentionally push the ball towards its own net when it was trying to move around the ball. The robot may be trying to go around it to a setup position on the other side, while maintaining tight control of the ball and keep opponent robots from sweeping in and stealing the ball. A simple solution would be to change the setup position to be further away from the ball in such a case. In the UM 2003 RoboCup program [10], this change would require additional time (more than necessary had the system made relevant information explicit and intuitive to work with) to implement and debug. This additional time may be crucial in many applications. If this was a larger modification, then the amount of time may be truly unacceptable. In the face of such changes, designing part of the system – specifically the part that may require frequent modifications – using a flexible and explicit representation (e.g. in a language based in XML) would prevent such disadvantages.

Furthermore, the developer needs to be aware of subtle nuances in the program; such as if two behaviours were both equally applicable to the current situation, then the first behaviour that was loaded would be used. This knowledge is made implicit in the C++ code and is not very intuitive for new developers.

As another example, in one game the defenders would not clear the ball (push the ball to the sides of the field and away from its own goal area). Subtle restrictions were discovered from the code of the *ClearBall* behaviour (the behaviour responsible

for clearing the ball). Here, the behaviour prevented the robot from clearing the ball unless it was very near to the goal. These constraints existed because the behaviour was initially coded for the goalkeeper. These constraints made sense for the goalkeeper robot (the goalkeeper is not supposed to leave the goal area), but did not for the other defenders. As a solution to the previous problem, a new *DefenderClearBall* behaviour needed to be created for the defenders; one that is free from those restrictions and allowed the robot to push the ball to the sides of the field away from its own goal area. However, this process relied on writing additional C++ code.

From the UM RoboCup teams' experiences at the robotics competitions, it was clear that an architecture was needed, and the requirements for this new architecture are as follows:

1. the process of designing behaviours needs to be simple and intuitive;
2. the goals and conditions of each behaviour need to be made explicit;
3. the process of creating and removing behaviours needs to be simple and flexible;
4. implementation aspects of the decision-making mechanism that chooses and switches between behaviours also needs to be made explicit.

Even though robotic soccer demonstrates real-world problems and highlights the need for powerful architectures, the resulting research is applicable to all intelligent mobile robot domains.

The following chapter provides background information on robotics, different control architectures (including behaviour-based ones), and other relevant information.

Chapter 2

Related Work

The research field of robotics is an interesting and unique field in Artificial Intelligence (A.I.) and Computer Science. It is a very practical domain, where theories and ideas are put to the test in real world physical environments. Also, not all the solutions are guaranteed to work precisely as expected all the time because there are too many factors to consider and too many unexpected situations can occur. Thus, the perfect or best solution is often unknown.

Designing an intelligent control program for a robot is a complex task. Agent architecture are used to structure the program to make it easier to debug, extend, and adapt. The terminology in the field has changed in recent years to favour agent-based approaches. Older systems can be modified to follow new agent-based approaches. Today, agent-based designs are used more often because of the advantages they provide. An agent is an entity that can reason and act for itself or on behalf of another. Most agents are autonomous, which means they reason and act for themselves without external control. Using agents allows for a distributed approach, which is advanta-

geous because it is scalable and flexible. Adding extra agents to control more robots would be simpler than modifying a centralized system. The overall system is also more protected against system failures because control of the system is distributed among the agents. Failure of one agent has a limited negative effect on the overall systems.

There are several criteria that can be used to characterize architectures [15].

- Domain Relevance versus Domain Independence;
- Analysis versus Synthesis;
- Top-down (knowledge-driven) versus Bottom-up (data-driven) design;
- Deliberative versus Reactive approaches.

The rest of this chapter is divided into sections (Section 2.1 – Section 2.4) based on these criteria.

2.1 Domain Relevance vs. Domain Independence

Domain relevance versus domain independence relates to the practical application of the architecture. Domain relevant architectures are strongly associated with the intended domain. This approach to designing an architecture is usually more efficient for tasks associated with the intended domain. It allows the developer to make assumptions about the tasks based on domain knowledge. For example, a chess-playing robot is not under as tight a real-time constraint as a soccer-playing robot. That is, it can take a little more time moving the pieces. As another example, a

seed-planting robot does not need as strict reactive control or obstacle avoidance behaviours as a vacuuming robot if it can assume it is working in a large open field.

Nevertheless, often the domain relevance directly ties the robot to the domain making the architecture difficult to extend into other domains. The opposite of the domain relevant approach is the domain independent approach. These architectures make little or no assumptions about the domain, hence they are more flexible. However, they often sacrifice some efficiency. That is, because they make little or no assumptions, they often must perform additional checks to enforce additional constraints on the behaviours. For example, an office floor-sweeping robot using the domain relevant approach may know that it only has ninety degree turns and operates at night with few dynamic obstacles. However, if a domain independent approach was used to design the architecture, the robot will be trying to optimize turns by a few degrees (e.g. eighty-five degrees vs. ninety-degrees) and perform a more intensive obstacle avoidance behaviour than would truly be necessary.

The issues relating to domain relevance versus domain independence in robot architectures are similar to issues relating to *"strong versus weak"* methods in artificial intelligence [15, 84].

2.2 Analysis vs. Synthesis

Analysis versus synthesis methodologies relate to assumptions about the very definition of intelligence. In analytical approaches, the idea of intelligence is abstracted into pieces based on the properties of an existing system. These abstract pieces are often inspired by intelligent behaviours of biological entities (e.g. dogs, humans, etc.).

Figure 2.1: Sony's Entertainment Robot, Aibo, from [7].

An example of this approach can be seen in the work of Arkin, Fujita, Takagi, and Hasegawa, who developed an architecture for Sony's Entertainment Robot, the Aibo (see Figure 2.1), based on the principles of *ethological modelling* (studying and modelling animals in their natural environments) [20]. The architecture they developed was based on popular studies of *canis familiaris*, the domestic dog. They created twelve subsystems that mimic the behaviours of a dog, which governs the actions of the robot. These subsystems (see Figure 2.2 for subsystem interaction model) are:

- Investigative (searching/seeeking);
- Sexual;
- Epimeletic (care and attention giving);
- Eliminative (excretion and urination);
- Et-epimelectic (attention getting or care soliciting);

Figure 2.2: Subsystems of the ethology architecture from [20].

- Ingestive (food and liquids);
- Allelomimetic (doing what others in a group do);
- Comfort-seeking (shelter-seeking);
- Agonistic (associated with conflict);
- Miscellaneous Motor;
- Play;
- Maladaptive.

This architecture uses a mechanism based on competition between behaviours for overall action selection. Behaviours uses two components to compete with other behaviours: motivation and interest levels. Inhibition is the opposing factor that suppresses each behaviour, which is applied iteratively until only one behaviour has a non-zero value. This behaviour will then become the primary behaviour. Other behaviours can execute as well provided that they do not interfere with the primary behaviour.

Overall, their architecture works well because they have a model of the level of intelligence they want to achieve. That is, they worked backwards from the goal to create their system. However, it is difficult to expand their system to achieve higher levels of intelligence than its current performance. For example, it is difficult to scale their system to model human intelligence. Also, proactive/goal-oriented tasks are often difficult to model in their system. Due to these problems, their architecture is not very flexible for use in other domain and tasks nor is it very adaptable in terms of adding more sensors, actuators, etc. As far as their architecture being intuitive, the developer must be knowledgeable with the behavioural aspects of a dog in order to work with this architecture. This is not very intuitive since the behaviours are not explicit in their representation.

Structure-in-5 architecture

Another example of the analysis approach can be seen in the work of Kolp, Giorgini, and Mylopoulos, in which they proposed a multi-agent architecture based on organizational theory [63]. Their architectural design, called *structure-in-5*, was

Figure 2.3: Overview of Kolp et al.'s architecture from [63].

modelled after a typical business organization model (see Figure 2.3). Nodes in their graphs were designated roles as *dependers* and *dependees* in which the *dependers* depend on the *dependees* for support. Their proposed architecture uses multi-agent research to highly modularize the components required to control a physical robot. In theory, this makes their architecture highly flexible and adaptable. However, their work has not been tested on a physical robot, and thus considerations for real-time performance seems to be lacking. Also, the design of their architecture seems overly complex and the behaviour representation is implicit in its design. This is not very

intuitive for new programmers. Thus, this architecture is not as useful as it could have been.

Synthesis approach

The synthesis approach is based on the idea that intelligence can be reduced to a single atomic unit, which when replicated and organized properly, can create high-level intelligent behaviour [15]. Synthesis approaches are akin to *unified field theory* approaches, which Arkin describes as employing the assertion that a single construct, when replicated, is sufficient to ultimately reproduce real-time human-level intelligence [15]. The distinction between these architectures primarily lies in what constitutes this single construct. In Brooks' subsumption architecture, this single construct is represented by a construct called a behaviour [28]. These behaviours are arranged in a layered fashion so that more complex and sophisticated behaviours can be created. Brooks' assertion is that with enough of these layers, real-time high-level intelligent behaviours can be reproduced [28]. Further discussion regarding Brooks' subsumption architecture is found later in this section.

RCS architecture

Another popular architecture which applies unified field theory to intelligence was developed by James Albus [8] for the Intelligent Systems Division (ISD) of the National Institute of Standards and Technology (NIST) [5] called the Real-Time Control System (RCS) [8, 54]. This architecture is similar to Brooks' subsumption architecture [28]. Instead of layers, it has multiple *levels*, which form a hierarchy. This may

Figure 2.4: Primary components of RCS architecture, taken from [8].

seem as a trivial difference, but there are certain features that set them apart. Each level has its own sensor processor (SP) module, world modelling (WM), value judgement (VJ) module, and behaviour generator (BG) (see Figure 2.4 for relationship diagram between components). The sensory processing component evaluates the perceptual data and compares it with the internal world model: that is, it can help filter some of the noise in the perceptual data. The world modelling component stores a database of knowledge representing the best estimate of the known world. The value judgement model makes decisions about actions that were deemed "good" and "bad", determines risk and uncertainty factors, and the attractiveness and repulsiveness of objects and regions of space in the world. The behaviour generation component takes

the goals of the system and generates plans to obtain those goals. The most attractive plan will then be executed. Sensor data moves up from level to level, world modelling information moves up and down the hierarchy, and behaviour generator commands are communicated down the hierarchy. Using this hierarchy, Albus proposes that intelligent behaviours can be created.

The overall RCS architecture appears to be fairly flexible and adaptable (i.e. it has been implemented on several platforms and domains). However, the programmer does not appear to be able to change any of the behaviours in the behaviour generation component easily. Behaviours are not made explicit, nor is the method how the architecture deals with conflict resolution. For example, what needs to be done if two goals conflict with each other. Thus, it is not very intuitive in this respect.

2.3 Top-down vs. Bottom-up architectures

Arkin has described the top-down vs. bottom-up architecture as akin to the *scruffy/neat* dichotomy in artificial intelligence [15, 84]. According to Russell and Norvig, the *neat* approach emphasizes formal analysis based on mathematical rigor while the *scruffy* approach emphasizes experimentation and discovery [84]. The top-down approach is similar to *neat*, whereas the bottom-up approach is similar to *scruffy*. In Arkin's definition, a top-down is knowledge-driven and involves a "*formal analysis and characterization of the requisite knowledge that a system needs to possess to manifest intelligent robotic performance*", while bottom-up is closer to experimentation and discovery [15]. However, these statements suggest that it relates to how a developer approaches the development of the architecture from a software

engineering standpoint. An alternative definition is how the *system* approaches its problem solving. The following sections (Section 2.3.1– 2.3.3) will relate more to the latter.

The next sections will further discuss the distinction between top-down architectures (Section 2.3.1), bottom-up architectures (Section 2.3.2), and hybrid architectures (Section 2.3.3).

2.3.1 Top-down architectures

A top-down architecture (see Figure 2.5) uses abstraction to decompose the perception, reasoning, and execution cycle. The motivation for the top-down architecture is that abstraction can hide details of the lower levels from the higher levels. For example, in theory a general path planning algorithm can be developed without the need to know about the locomotion capabilities (e.g. wheels, legs, or snake-like motion) of the robot. In practice, the two levels do interact in subtle ways. The reasoning system cannot be developed independently of the sensors and vice versa. For example, there are two approaches to global vision in robotic soccer. Most teams mount the camera directly over the playing field, whereas our team (The RoboBisons) [10, 11, 12] uses a oblique view [23]. In theory, the exact position and view of the camera should not affect a behaviour such as dribbling the ball. In practice, since with a side mounted camera, the robot will occasionally occlude the ball, the *dribble ball* behaviour must back off the robot from the ball periodically to make sure that the ball is still in front of the robot. This backing off is not necessary with a camera mounted directly overhead.

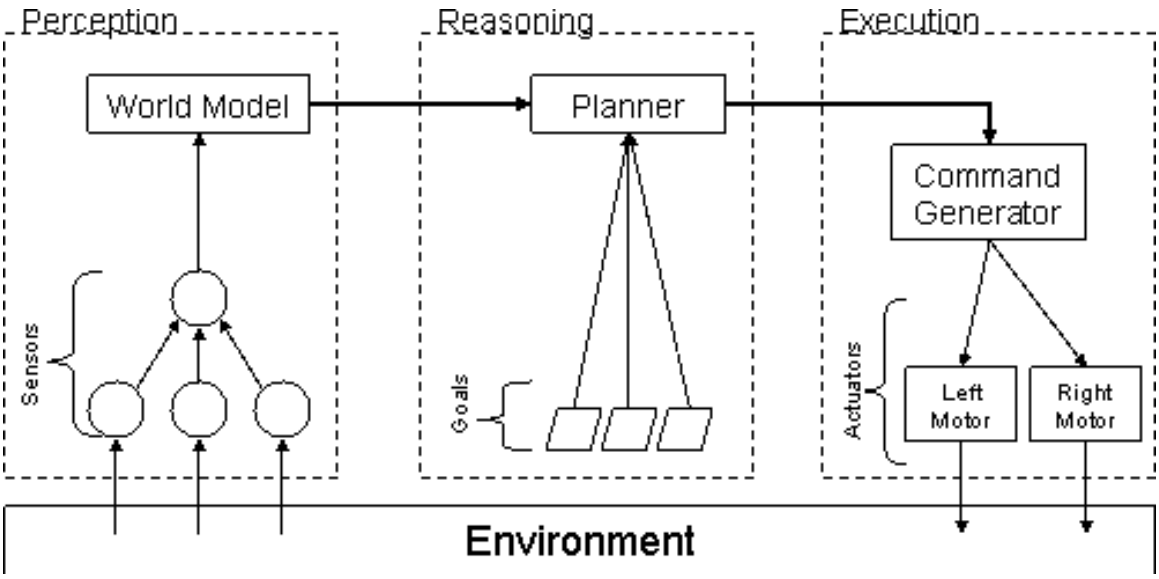


Figure 2.5: Generic Top-down Architecture.

Pure top-down architectures have an explicit world model and focus on devising one strategy and carrying it through to the end. These systems are good at planning and higher level reasoning, but are not reactive enough for dynamic environments. To overcome this problem, researchers have developed extensions to pure top-down architectures, such as behaviour trees. A behaviour tree is a collection of behaviours organized in a tree [40]. It maps complex behaviours by branching them into smaller simpler behaviours. The depth of the tree depends on the complexity of the most complex behaviour, while the breadth of the tree depends on the number of behaviours. Behaviour trees are very useful to help manage the complexity of one branch from another on the same level. However, it is difficult to jump from one state in a certain level in a branch to a different state in another level of a different branch.

For example, in the soccer domain, assume a robot is performing a passing behaviour maneuver that requires it to move to a specific location before it can pass (see

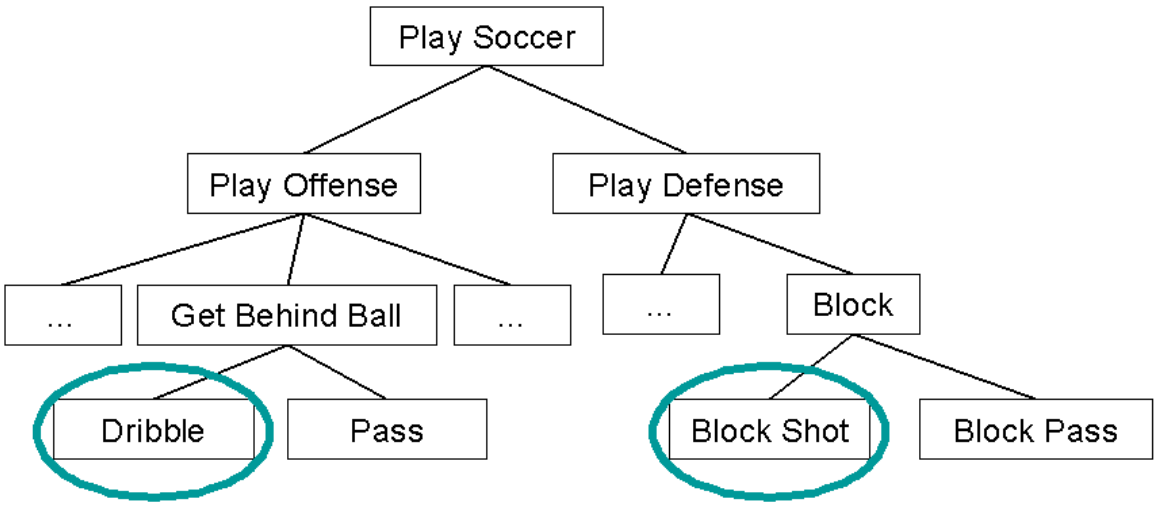


Figure 2.6: Simple Behaviour Tree.

Figure 2.6). Due to the complexity of this particular maneuver, the behaviour can be nested several layers down the tree. If the robot suddenly loses the ball, and now has to block the opponent robot, the robot system logic must move up a certain number of levels and then down another number of levels until it reaches the state that has the appropriate behaviour to perform. Quite often, moving back up is not possible, and it will have to derive the new appropriate behaviour by starting from the top of the tree and working down. This is important when it comes to improvising or coping with unexpected events. Also, behaviour tree systems must often deal with large computational complexity, and therefore perform too slowly to work efficiently in dynamic real life environments.

2.3.2 Bottom-up architectures

Bottom-up architectures (see Figure 2.7) are the opposite of top-down architectures. Instead of multiple levels of abstraction in the perception, reasoning, and

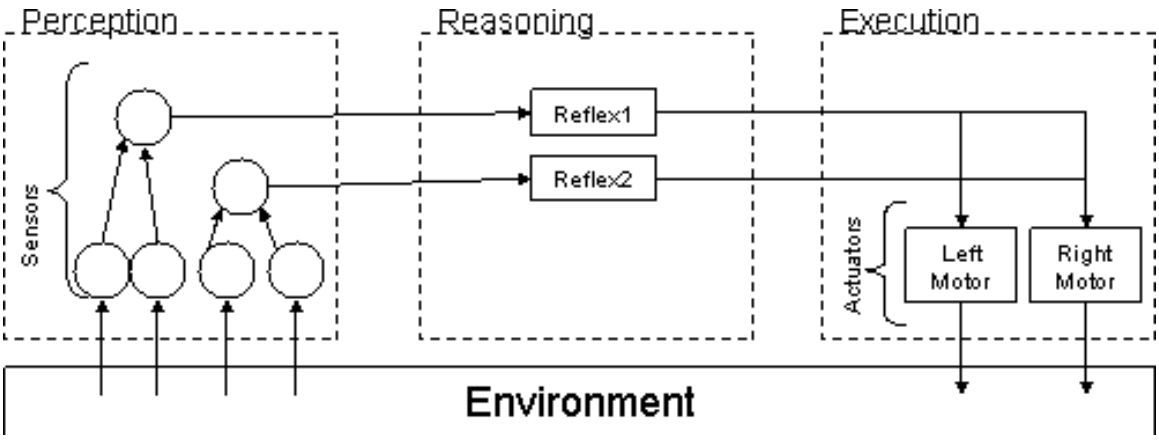


Figure 2.7: Generic Bottom-Up Architecture.

execution stages, bottom up architectures include simple behaviours that map perceptions directly to actuator commands (similar to reflexes). More complex behaviours are created by combining simpler ones. Bottom-up architectures are able to react quickly to the environment because of the direct links between sensors and actuators (e.g. avoid an obstacle). The disadvantage of bottom-up architectures is that it is often difficult to know what lower level behaviours are needed and to predict the interaction of multiple behaviours. Some bottom-up architecture systems include reactive architectures and Brooks' subsumption architecture [28].

Pure reactive architectures have a set of low-level behaviours which can react to certain situations when they arise. The main advantage of using these systems is that they are computationally efficient. This is a desired quality when working in dynamic environments such as robotic soccer. However, there are disadvantages of such systems as well. The main disadvantage is that these systems usually have no mechanism for higher level planning or reasoning. In other words, they are not

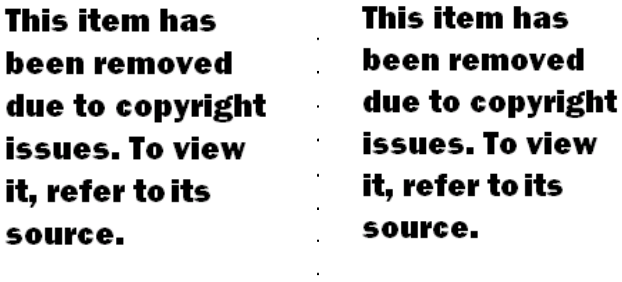


Figure 2.8: Brooks' layering concept, taken from [28].

proactive and often require an event to trigger any behaviour. Furthermore, it is often difficult to generate or select among several goals.

Brooks' Subsumption Architecture

Brooks' subsumption architecture is one well-known architecture for controlling robots and their behaviours [28, 29]. This architecture is one of the most studied architectures by many researchers [50, 92]. It does not use an explicit world model like pure top-down systems. The main concept of Brooks' subsumption architecture is that the robots behaviours are designed in a layered approach (see Fig 2.8). Each layer is an asynchronous module and higher-level layers have the ability to *subsume* (i.e. override) the lower layers. The higher layer subsumes the lower layers by either inhibiting the inputs to or suppressing the outputs of the lower layers. This makes the architecture robust when new or additional behaviours are required. Gorton and Mikhak demonstrated how the subsumption architecture can be easily integrated into hardware (for performance) of a simple platform (see Figure 2.9, 2.10, and 2.11) [50].

Figure 2.9: Gorton and Mikhak's robot platform, taken from [50].

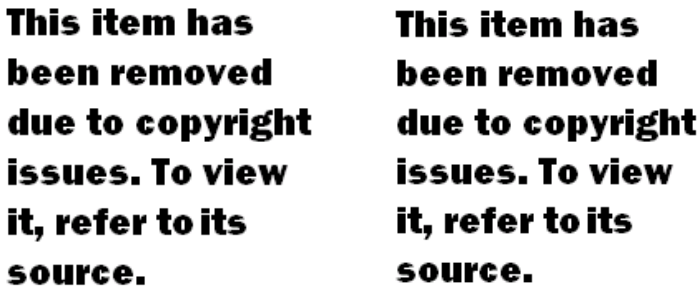


Figure 2.10: Gorton and Mikhak's architecture, taken from [50].

Brooks' subsumption architecture is very flexible and adaptable and has also been used successfully in robotic soccer [82, 83]. Nevertheless, there are several disadvantages with Brooks' subsumption architecture. The major disadvantage is that the complexity of designing the system increases greatly as more complex higher level layers are added. Thus, it is limited in its extensibility and also in its intuitiveness for developers.

Figure 2.11: Gorton and Mikhak's hardware prototype, taken from [50].

Rocky III

Miller, Desai, Gat, Ivlev, and Loch developed a robot called Rocky III, based on a reactive, bottom-up architecture [74] and programmed using the ALFA language [46]. Their architecture was slightly different from Brooks' Subsumption architecture [28]. Instead of higher layers subsuming the function of the lower layers, the higher layers provide information to the lower layers. Their architecture is a three tiered architecture. The lowest level interfaces with the sensors, and actuators. The second level computes the motion required to pass to the lower level. The highest layer consists of the master sequencer which basically contains a list of waypoints for the robot. Their architecture is fairly simple and has been used only for reactive navigation. However, it is not flexible enough to change a particular layer or to add more task-directed behaviours, such as performing passing plays with time constraints in the robotic

soccer domain. The disadvantages of this architecture are similar to those of Brooks' subsumption architecture mentioned previously.

Bottom-up architectures are fast and efficient, but hard to maintain since it is difficult to predict how adding in a reflex changes the behaviour of the system. For example, changing a sensor can affect the validity of obstacle avoidance behaviours.

2.3.3 Hybrid architectures

Hybrid architectures are a mix between top-down and bottom-up architectures. These types of architectures are the most popular because they take advantage of the strengths of top-down and bottom-up architectures [9, 19, 26, 34, 36, 38, 47, 73, 75, 88, 89]. However, because they also inherit the weaknesses of those architectures, the difficulty lies in finding a reasonable balance between the two types of architectures.

In hybrid architectures, some sensors and perceptions are directly connected to the actuators, whereas others are processed more extensively. Instead of a complete world model, the perceptions place the system in a finite set of states. In other words, the environment can be mapped to certain states in the system. The reasoning system moves the agent into desired states. For example, the *kick to goal* behaviour includes three states (position behind ball, facing the ball, and kicking the ball into the goal). One main advantage of using such an architecture is that it is easy to transfer from one state to another on the same level (compared to that of top-down architectures). For example, if after approaching the ball the robot is already facing the ball, then the system will kick the ball immediately.

Hierarchically focus vs. Non-hierarchically focus

The popularity of hybrid architectures results from the belief that a combination of planning and reactive architectures will realize the benefits of both. However, hybrid architectures can be further categorized by hierarchically focused and non-hierarchically focused. Hierarchically focused architectures tend to focus on breaking down the problem of intelligent autonomous robots into layers, levels, or tiers. Quite often, lower levels tend to focus more on specific implementation consideration problems such as control and sensory data management. Higher levels focus more on general abstract problems such as planning, task management, or task sequencing. With regard to non-hierarchically focused architectures, it is not that the distinction between levels is completely absent, but rather that the layered aspect is not the most important part of the architecture. Other aspects such as parallelism (having more than one component execute in parallel and intelligent behaviours emerges as the result of the interaction between these components) may be emphasized more than the hierarchical aspect. The following example architectures elaborate on this point.

ATLANTIS architecture

Gat proposed an architecture called ATLANTIS, which combines planning and reacting [47]. He claims that his architecture is capable of producing behaviour which is reliable, task-directed and reactive to unexpected situations. ATLANTIS is based on a continuous action model, which means that the operators consume negligible amount of time. These operators, called *decisions*, do not control the robot directly. Instead, they initiates processes called activities, which control the robot. But because

there is no strict relationship between the decisions and the changes in the world, this activity action model is more difficult to analyze compared to state-based action models.

The core principles of ATLANTIS, are based on three components:

1. a controller - a reactive control mechanism that controls primitive activities, which does not require decision-making computations;

2. a sequencer - a mechanism to control the initiation and termination of the activities in the controller;

3. a deliberator - performs time-consuming planning activities and world modelling.

The controller in ATLANTIS uses the ALFA programming language [46]. ALFA was used to create a framework for controlling the robot, instead of using classical control theory, due to the difficulty of constructing and adequate mathematical model. Using ALFA allowed the programmer to easily describe the functionality, and thus the implementation of the hardware.

The sequencer in ATLANTIS is based on the Reactive Action Package (RAP) system [44]. In general, the sequencer is designed to almost never fail, or rather failures are assumed to be rare occurrences. This means that the sequencer is limited in its capabilities of dealing with failures. In complex dynamic domains such as robotic soccer, quite often a sequences of activities must be interrupted because the world is constantly changing. Not only are these activities interrupted, but they are not necessary.

The planner in his architecture is a traditional symbolic planner, which is supposed to *guide* the robot and not directly control it. This alleviates much of the burden from the programmer to implement the interface between the planner and the sequencer. However, the drawback is that it is more difficult to evaluate. Overall, this architecture is not very intuitive. Also, his architecture has been only used in the task of robotic navigation, which means that it is not very flexible and extensible.

SSS architecture

Connell's SSS architecture is another hybrid architecture [36] which was developed for robot navigation in mind. It is primarily based on three layers (see Figure 2.12): the Servo layer, Subsumption layer, and Symbolic layer (hence the "SSS" acronym for the name). The Servo layer works directly with the sensors and actuators. The Subsumption layer is supposed to recognize specific situations and assign setpoints (points to which the robot should move) to the servo layer, which will activate the appropriate actuator to perform the desired motion. The Symbolic layer sets the parameters to the subsumption layer and receives feedback from the Subsumption layer to know if the parameters are correct. A special construct of this architecture is the "contingency table", which is used to relieve some real-time burden from the symbolic system. The contingency table continuously monitors a collection of special purpose situation recognizers. When the situations/events of interest occur, the contingency table quickly passes a new set of parameters to the subsumption layer.

According to Connell [36], behaviour-based/subsumption systems do not impose as many modelling constraints on the world and are good at making rapid radical de-

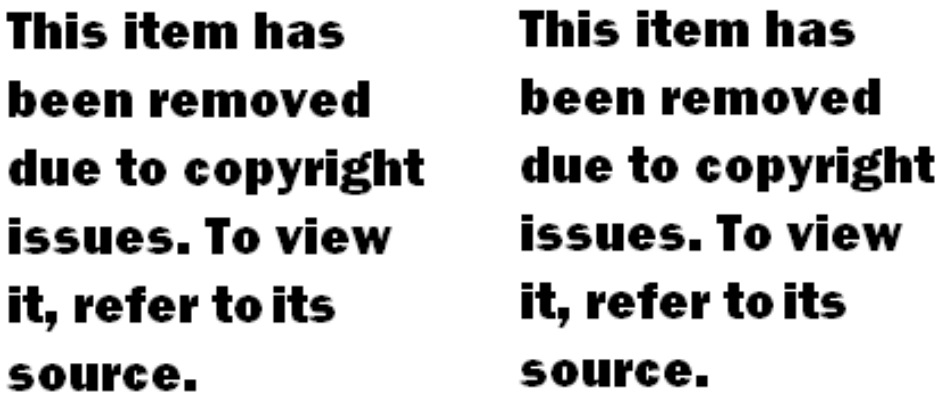


Figure 2.12: Diagram of the SSS architecture, taken from [36].

cisions. However, behaviour-based systems often have problems with world modelling and persistent state. This is because behaviour-based systems are often developed with a distributed design in mind so it is difficult to find a suitable location to store this information.

There are a few drawbacks in the SSS architecture. First using a symbolic language at the top layer is flexible, but it does reduce performance. This is even apparent to Connell, as he uses the contingency table as a fix. Secondly, the contingency table solution is a somewhat non-intuitive solution, and is not really flexible enough for complex dynamic domains.

3T architecture

Bonasso et al. developed an architecture called the 3T robot architecture [26]. The 3T architecture has been used on robots for simple office hallway navigation, trash-

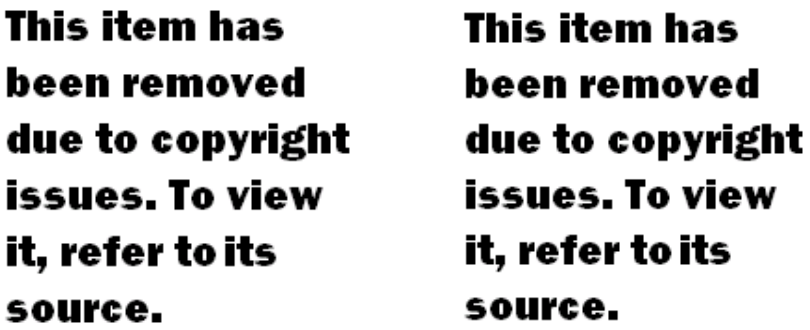


Figure 2.13: Diagram of the 3T architecture, taken from [26].

collecting robots, and even robots that work on space stations. Their architecture separates the robot intelligence into three *tiers* (hence the name) (see Figure 2.13). The three parts are:

- a dynamically reprogrammable set of reactive skills coordinated by a skill manager;
- a sequencer that controls the activation of skills in the system;
- a deliberative planner that reasons about the goals and constraints.

The dynamically reprogrammable set of reactive skills are called *situated skills*. Situated skills represent the 3T architecture's connection with the world [87]. That is, they are configurations of the robot's control system that will achieve or maintain a particular state in the world. They are called *situated* because the proper context must be present for the skill to be effective. For example, a robot may have an arm to lift objects, however if the robot is placed in an empty room devoid of any movable objects, this skill is not useful. Thus, knowledge of when the skill is applicable is

important. The situated skill has certain specification components to ensure proper usage: inputs and outputs, initialization, enable and disable components, and computational transform. The inputs and outputs provide the preconditions for the skill and links to the following skills. Initialization performs any starting configuration that is necessary to perform this skill. The *enable and disable* components allow or disallow the skill to function. Finally, the computational transform implements the skill's functionality.

The sequencer organizes the skills for activation. This system is reasonably simple in that it places the skills in discrete steps that must be performed. The sequencer used in the 3T architecture is based on Firby's RAP system [44].

The highest layer, the deliberative component, was placed as high as possible to reduce the amount of processing that the deliberative component is required to perform. Their planner has some attractive features such being designed for multi-agent coordination, and it can also reason about agents that the system does not control.

The three tiered approach of the 3T has been used in many other architectures because of the way it modularizes the problem of autonomous intelligent robots. Compared to the SSS architecture, which is also three tiered, the 3T architecture is more flexible for certain tasks. In practice, the SSS architecture has only been demonstrated with pure navigation. Nevertheless, because the behaviour components are implicit, it is not very intuitive.

NASREM architecture

NASREM is an architecture that predates 3T, but shares many of the same features [9]. It is a multi-tiered architecture, which provides several levels of abstraction. One of the primary difference between NASREM and 3T is that 3T maintains a global world model. NASREM is also mainly a reference model, and not an implementation. Its disadvantages are similar to those of the 3T architecture.

Saridis' architecture

Saridis proposed another architecture for intelligent control [85]. This architecture also has three layers. Saridis takes a more bottom up approach in the design of his architecture. The lowest level starts with the servo systems available and modifies them for use for the level above. The next level consists of coordination routines for the lower subsystems, and a scheduling mechanism which is structured in a network-like manner. The highest level uses a neural net to find a sequence of actions that matches the text input. However, this architecture does not handle failure (on each of the three layers) well and is not as dynamically reconfigurable as architectures such as the 3T. Thus, it is even less flexible and intuitive than the 3T architecture.

Noreils and Chatila's architecture

Noreils and Chatila proposed another three tiered architecture [76]. The three tiers are planning, control, and functionality. The control level of this architecture distinguishes between failures and successes. In comparison with architectures such as the 3T architecture, which does not make this distinction, this may be useful. For

small tasks, this is useful in the short run to allow the planner to know which actions will be beneficial. However, in larger tasks, situations will not fit into these two discrete, crisp categories: rather, they are fuzzy in nature. Problems such as these can be seen in classical A.I. problems such as the game of chess. It is often difficult early on to know if choices for actions are beneficial or not. The disadvantages of this architecture is similar to those of the 3T, Saridis' architecture and related architectures.

Ranganatha and Koenig's architecture

Ranganatha and Koenig proposed another hybrid architecture [79]. The base of this architecture is reactive in nature. However a deliberative component is given progressively greater control over the robot in situations where the reactive component fails to make any progress. Their architecture is also a three-layered architecture with the following layers: reactive layer, sequencing layer, and a deliberative layer. The reactive layer uses motor schemata [14] in the form of two simple behaviours: move to some given coordinates and avoid obstacles. The deliberative layer performs high level path planning. The sequencing layer evaluates the progress of the reactive layer. If not enough progress has been made (e.g. the robot is stuck in a local minima/maxima), then the deliberative layer is allowed to control the robot using a waypoint. Their architecture uses three different modes of control:

1. mode 1 - reactive layer has full control;

2. mode 2 - reactive layer has most of the control using the advice (a waypoint) provided by the deliberative layer;

3. mode 3 - the deliberative layer has full control since the reactive layer is not performing or did not perform well given the terrain.

Their architecture provides an interesting view of hybrid architectures. Nevertheless, their architecture is adequate for the domain of robotic navigation. However, it is insufficient for more goal-directed behaviours because the only method for the deliberative layer to control the robot is via a waypoint rather than a more broad means. Thus, it is not very flexible. Also, the difficulty of modifying and maintaining such an architecture is unclear.

LICA architecture

Hu, Brady, Grothusen, Li, and Probert proposed another hybrid agent-based (meaning some components of the architecture are agents) architecture called LICAs (Locally Intelligent Control Agents) [53]. A strong feature of their architecture is that control tasks are distributed among agents called *behaviour experts*. These behaviour experts tightly couple sensing and action, and they also execute in parallel in a fashion similar to Brook's subsumption architecture [28]. However, these behaviour experts are loosely coupled in a hierarchical manner. The design of a LICA module is shown in Figure 2.14. The layers execute in parallel and the lower layers execute faster due to the simpler processing required. This architecture is as flexible as Brooks' subsumption architecture. However, some aspects are not defined, such as the task and behaviour selection, as well as conflict resolution. For example, consider a scenario where one module is sensing and directing the robot to head one direction, while another module is directing the robot to head in another direction. The mech-

Figure 2.14: LICA software configuration diagram, from [53].

anism to resolve this problem is not explicit. Without this being made more explicit, this architecture is not intuitive for debugging, testing, or modifying.

Non-hierarchically focused architectures

The following architectures are other hybrid architectures, however they do not focus on using a layered approach to bridge the gap between deliberative and reactive architectures. Although they may have some hierarchical components, it is not those components themselves that are the essence of the architecture. For example, the AuRA architecture by Arkin [16]) has a hierarchical planner, but it is essentially

only one half of the architecture itself. The other half is a reactive schema-based component.

AuRA architecture

AuRA (short for Autonomous Robot Architecture) is one of the most note-worthy architectures [16, 19]. AuRA is a hybridization of a deliberative hierarchical planner and a reactive controller. It was the first robotic navigation system to present this style of integration [17]. The components of the AuRA architecture are shown in Figure 2.15. The hierarchical component consists of the mission planner, spatial reasoner, and the plan sequencer. The mission planner was the interface for the human controller. The spatial reasoner, also known as the navigator, helps plan the paths the robot must take to complete the missions assigned. The plan sequencer translates the path segments into motor commands and queues them for execution. The schema controller or schema manager uses schemas (based on schema theory [13]), which are mappings of perceptions to motion commands. Some of the strengths of the AuRA architecture are that it is highly modular and flexible, and it provides for additional adaptation and learning methods.

The AuRA architecture has been used successfully on numerous physical robots. Arkin and Mackenzie showed how the AuRA architecture can make use of *a priori* information to improve mobility [21]. Collins, Arkin, and Henshaw [35] demonstrated this on a robot called "Buzz" in the first robot exhibition sponsored by the American Association for Artificial Intelligence (AAAI) [2]. However, the emphasis was on the reactive component since it was sufficient enough for the specific task at hand.

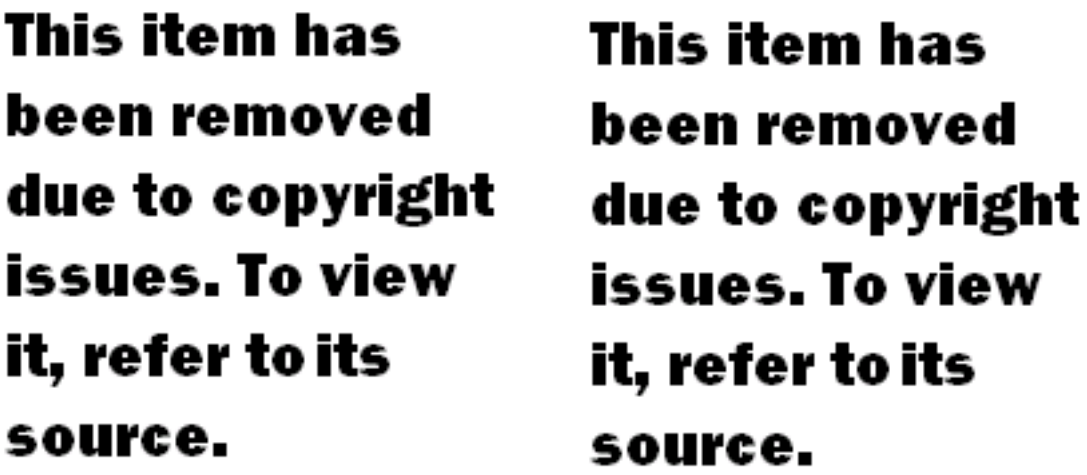


Figure 2.15: AuRA architecture diagram taken from [19].

Nevertheless, much of the behaviour mechanism is implicit in its design. Thus, it is not intuitive for new developers to modify and extend.

Stoytchev and Arkin's architecture

Stoytchev and Arkin proposed another robot architecture [89]. Their architecture was designed to combine deliberation, reactivity, and motivation in a behaviour-based system. This architecture differs from architectures such as the AuRA architecture because it adds a third component, specifically the motivation component. The deliberative component performs the path planning similar to other architectures. The path planner also uses a Finite State Automata (FSA) to sequence the path segments. The reactive component is based on schema theory [13]), similar to the AuRA archi-

tecture. There are two primary subsystems of this component: the process monitor, which is used to keep track of the progress relating to the current task and the exception manager, which is used to deal with unexpected situations. The motivation component is similar to other ethologically inspired architectures [20]. Motivation variables can contain values between 0 and 1 to describe the strength of the particular motive. The motivation with the highest value will dictate the general behaviour of the robot.

This architecture is useful for simple intelligence such as navigating office hallways. However, the flexibility of the reactive component is not robust. For example, it was not able to handle dead-reckoning errors. This is a significant weakness in this architecture as well as the AuRA architecture. Thus, these two architectures are not very adaptable and also not very intuitive.

Reflecs architecture

Goel, Stroulia, Chen and Rowland developed a hybrid architecture called Reflecs, which is, like the LICA architecture, agent-based [49]. Reflecs' main components are a schema-based reactive component and model-based deliberative reasoner. Redundancy and multiple configurations are possible due to the schema-based reactive component. The deliberative reasoner is part of an agent which is able to detect failure (i.e. local minima/maxima). The deliberative reasoner uses a Structure-Behaviour-Function (SBF) model. This model uses three kinds of knowledge.

1. Functions and modes of the perceptual and motor schemas;

2. The tasks (or Functions), the methods of performing those tasks (or Behaviours), and the primitive schemas structure (Structure);

3. Redundancies in the design, thus allowing the system to switch modes/behaviours.

This is another example of an architecture whose primary system is the reactive component, with the deliberative component used mainly as a backup. Their design is very flexible. However it is not very intuitive because the behaviours are implicit in their design.

Soldo's architecture

Soldo proposed a hybrid architecture with reactive and preplanned control in a mobile robot [88]. His architecture is also agent-based as the control of the robot is distributed among a set of *behaviour experts* (see Figure 2.16). These behaviour experts perform some specialized processing on the sensor input, which then adjusts the robot state and the world state. Based on the new robot and world state, additional specialized processing is performed to create a robot action. In summary, a behaviour expert maps sensations to a particular action. Soldo states a set of these experts create the overall *behaviour* of the robot (see Figure 2.17). There are also *boundary experts* (daemons) whose function is to trigger changes in the overall behaviour. These boundary experts provide the preplanned control, while regular behaviour experts provide more reactive control. Also, Soldo's architecture does not use a pre-stored map, but performs some map generation dynamically as it identifies "landmarks" in the world. Goal-directed behaviour requires a map. This implies

Figure 2.16: Soldo's Behaviour Expert diagram from [88].

Figure 2.17: Soldo's Robot Behaviour diagram from [88].

much of the overall behaviour is reactive until a map is well-formed, to the point where it can be useful to meet the robot's objectives.

There are several disadvantages to this architecture. First, Soldo's merging of sensing and action into behaviour is not very adaptable. It is true that tighter coupling provides improved performance, but if those sensors were to fail, then those behaviours would not be very useful if they cannot work with other sensors. Making modifications to work with new sensors will take a long time if this modification is needed for all behaviours in the system. Thus, it is not very flexible. Secondly, behaviours may conflict with each other. Soldo does not specify a mechanism to deal with conflict resolution. This is not intuitive for new developers.

Michaud, Lachiver, and Dinh's architecture

Michaud, Lachiver, and Dinh also worked on developing a new hybrid architecture, which combines reactivity, planning, deliberation and motivation [73] (see Figure 2.18). Their architecture uses an agent-based approach, and the basic control component is called a behaviour. The external situation module, the needs module, the cognition module, the motives module, the behavioural module and the final selection module form the six primary modules in their proposed architecture. Behaviours all run in parallel and are selected dynamically depending on the intentions of the agent. Their resulting commands are blended together based on their respective importance. This importance level is determined by the external situation module, the needs module and the cognition module. The external situation module evaluates the robot's environmental conditions that affect behaviour selection (e.g. obstacles). The needs module selects behaviours based on the robot's needs and goals. The cognition module is for cognitive recommendation based on learning how the agent works (its reactions and behaviour selections of the past) in the environment. The recommendations of these three modules are given to the final selection module, which activates the selected behaviour. The motives module contains motives, which coordinate the operations of the other modules.

However, their architecture does not allow for abstraction of behaviours. This abstraction is useful for developers to incrementally design their system. Using an abstraction feature would have made their system more flexible. So far, their architecture has also only been tested in simulation. It is difficult to know how well their architecture would scale up to real world robots. Their architecture is somewhat in-

Figure 2.18: A diagram of Michaud et al.'s architecture, taken from [73].

tuitive already, since it has the behaviour separated and specifies how the behaviours are to interact. However, it can be more intuitive if the behaviour design used a specification language.

TCA architecture

Simmons proposed an architecture called Task Control Architecture (TCA) [86]. His architecture uses a task net or task tree, which can be further decomposed to simpler, lower-level control and functionality. These task trees do not have a specific representation, which increases the flexibility of this architecture. However these task trees are manipulated directly by C function calls, which the designer is responsible for. It also uses a sophisticated and complicated message-passing algorithm, and a

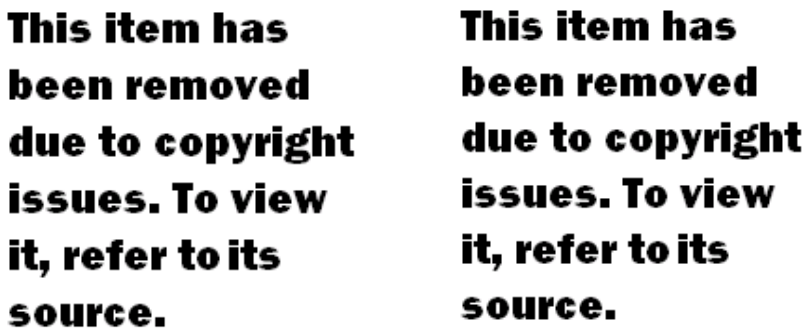


Figure 2.19: A diagram of Low et al.'s architecture, taken from [70].

central router to handle these messages. Thus, it is not very intuitive for developers to modify and extend.

Low, Leow, and Ang's architecture

Low, Leow, and Ang proposed an hybrid architecture that integrates deliberative planning and reactive behaviour-based control (see Figure 2.19) [70]. Their architecture has two key features:

- The planning module produces a sequence of checkpoints instead of a conventional path;
- The reactive module also uses a self-organizing neural network for control.

The deliberative component plans a path using cell decomposition similar to the method found in [78], which generates a series of waypoints/checkpoints. The reactive component uses the checkpoints as targets to be reached. Combining those targets with an obstacle avoidance module and a homeostatic control module to regulate the

robot's execution module, the high-level control system generates low-level commands to its actuators to move the robot. This hybrid architecture is well suited for the domain of robot navigation. However, the main drawback of this architecture is that manipulating the robot for another task is not easily done. For example, this architecture did not take into consideration temporal constraints, and therefore is not suited for any tasks that calls for it. Thus, this architecture is not very flexible. In addition, it is not very intuitive, as many important aspects are implicit in its design.

Nicolescu and Mataric's architecture

Nicolescu and Mataric proposed another interesting hybrid behaviour-based architecture [75]. Their architecture was developed as a result of considering two important issues lacking in pure behaviour-based architectures. First, pure behaviour-based architectures tend to rely too strongly on reaction and not enough on abstract representation, and second, that behaviors are generally designed by hand for one specific task and are not very reusable. Thus, their architecture employs *primitive behaviours*, which are tightly integrated with the lower level command generations that actually control the actuators, and *abstract behaviours*, which are more high-level. Abstract behaviours simply specify behaviour conditions, and thus can be concerned with higher level issues rather than lower level details (e.g. control) (see Figure 2.20 for a diagram of the behaviour relationship). In addition, there are also *Network Abstract Behaviours*, which represent a tree-like network of behaviours. This construct is used to abstract a large section of the overall behaviour tree. Abstracting large sections helps developers manage large complex behaviours systems. It gives

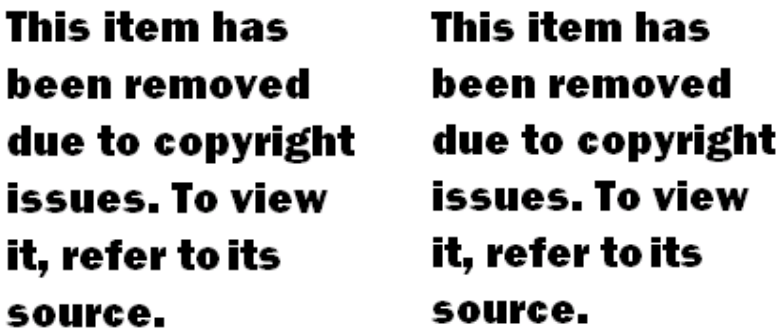


Figure 2.20: A diagram of Nicolescu and Mataric's behaviour architecture taken from [75].

the behaviour system the appearance of simplicity and readability from a high-level perspective. This increases their architecture's intuitiveness for new developers to work with (e.g. adding/changing/removing behaviours). From a reusability standpoint, this is also very useful. The developer can make use of an network abstract behaviour to perform the actions of the large behavioural section that the network abstract behaviour is supposed to represent. This can save the developer time from redeveloping these large sections.

Their architecture allows more flexibility compared to historical behaviour-based architectures. In addition, their architecture makes great use of abstraction to simplify behaviour representation and manageability. However, the design of the behaviours and other high-level specifics can be made even more intuitive for new programmers by using more explicit representation.

Potential fields based architecture

Laue and Röfer proposed another style of hybrid architecture [65], which is similar to the proposed architecture of this thesis. One interesting point to note is that their architecture uses competition instead of superposition between behaviours as means for action selection. Also, potential fields [18, 58] are tightly integrated into their behaviours (see Fig. 2.21); not only in the traditional sense as a means of obstacle avoidance, but also as part of the action selection process. Behaviours in their architecture also use explicit representation using XABSL [68], which makes their architecture more intuitive. Further discussion of their behaviour representation can be found in Section 2.5. Their architecture has been tested in the robotic soccer domain on Sony Aibo dogs [81] and the B-Smart small-size robots [69]. Thus, this architecture is somewhat flexible already. However, there are some drawbacks to their architecture, which is further discussed in the next chapter.

Decugis and Ferber architecture

Decugis and Ferber describes another architecture that uses competition as means for action selection [39]. Their architecture expands on the works of Maes [71, 72] by including a hierarchy of levels where a collection of flat scope behaviours resides. This makes their design flexible. However, the design of the behaviours themselves does not use explicit representation, and thus it is not very intuitive in this respect.

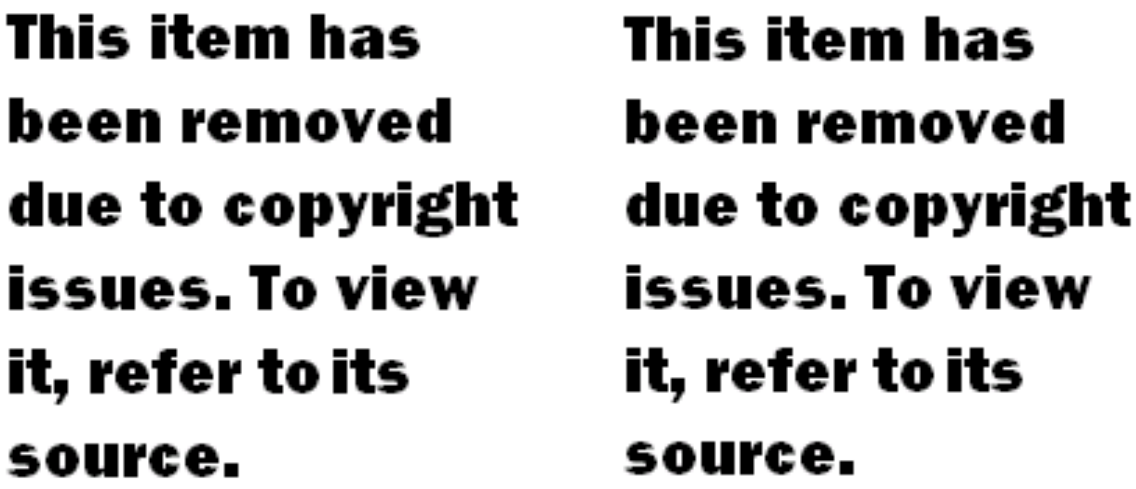


Figure 2.21: Extracts from an XML behavior specification from [65]: a) An object class describing the attributes of an opponent robot. b) A motion behavior for moving towards a ball.

Ubiquitous Robot architecture

Kim, Lee, and Kim proposed a unique idea of *artificial chromosomes* as the basic foundation of creating a ubiquitous robot [59, 60, 61]. Figure 2.22 shows three types of artificial genes. *F-genes* represent fundamental characteristics and genetic information such as sex, life span, color, and initial and mean values of internal states. *I-genes* represent internal preference settings (i.e. weights) on such matters such as internal state and external stimuli. *B-genes* represent the weight settings for behaviour selection, activation levels, and activation frequency. Using these genes, they created

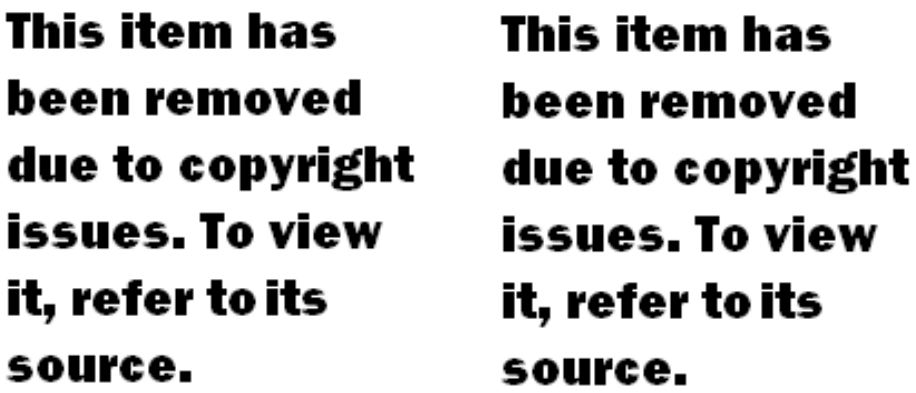


Figure 2.22: Artificial chromosome diagram, taken from [61].

a *Sobot* prototype (a software/simulation robot) called Rity. Figure 2.23 shows a diagram of the Sobot Rity's architecture. The goal was to incorporate a Sobot as part of a ubiquitous robot (Ubibot), a new generation of robot that exists in the virtual world and real world. From a high level perspective, the Ubibot is comprised of three parts: the Sobot, the Embot (embedded robot - the controller), and the Mobot (mobile robot platform) (see Figure 2.24). So far, their work has been implemented in simulation alone. Whether or not their architecture will work as planned remains to be seen. Thus, to this point it is unknown whether much consideration has been put forth regarding how to interface the Sobot intelligence and the actual physical hardware of an actual mobile robot. This could be problematic if such a design was an afterthought. For example, the physical robot may not support some behaviours or behaviours may have been too well trained to be modified. Thus, it may not be very adaptable in this respect. Also, modification of the behaviours would then be

Figure 2.23: Architecture of Rity diagram, taken from [59].

Figure 2.24: Ubibot diagram, taken from [59].

a very difficult task. Thus, it is not very flexible in this respect. Using gene representation is a novel idea – however, most developers are not biologists. Much of the relevant information is made implicit in their design, and thus it is not very intuitive for developers to work with.

Blackboard architectures

Blackboard architectures has also been proposed in the past for use with mobile robots [30, 62]. The advantages of blackboard-based architectures are:

- They are able to unite several knowledge sources (whether they are homogeneous or heterogeneous in nature) for data processing;
- They use simple and effective communication between experts (e.g. domain knowledge expert agents, control knowledge expert agents, etc.);
- They support the incremental construction of a solution (e.g. for path planning).

Two important blackboard-based architectures include Koenig and Crochon's TRAM architecture and Brussel et. al's blackboard-attention architecture.

TRAM architecture

Koenig and Crochon proposed an architecture called TRAM (short for Tableau-noir pour Robotique Autonome Mobile, in French) in 1988 [62]. There are four main components of the TRAM architecture: the domain blackboard, knowledge sources, the control, and the robot environment. The domain blackboard is a globally accessible blackboard or database that contains state information about the problem/task and the steps (taken or hypothesized) to solve the problem or complete the task. Knowledge sources contain information specific to an area of expertise (e.g. planning expert, or control expert, etc.). The control component manages the blackboard and the knowledge sources. This allows for a separation between domain information and control information. The robot environment is another globally accessible construct

similar to world state information in other architectures. Their architecture is somewhat flexible with their modular design. However, it is not very intuitive since much information (e.g. behaviour definitions) are not made explicit.

Blackboard-attention architecture

More recently, Brussel, Moreas, Zaatri, and Nuttin developed another behaviour-based blackboard architecture (see Figure 2.25) [30]. Their hypothesis in using the

Figure 2.25: A diagram of Brussell et al.'s blackboard architecture, taken from [30].

blackboard method was to overcome the lack of flexibility of hierarchical architectures, and the difficulties of state representation and integration of world knowledge. Their blackboard serves as a communication medium between components and provides information about state representation. In their blackboard architecture, the key feature is the concept of *attention*. Behaviours have different attention levels depending on the constraints associated with the behaviour and on different stimuli. More attentive behaviours will make more attempts to exert control over the actuators. The disadvantages of this architecture are similar to the previously mentioned blackboard architecture.

2.4 Deliberative vs. Reactive

Deliberative vs. reactive architectures relate more to the function of the architecture than to its structure or organization. Deliberative architectures have plans preprogrammed, or use a planner to create plans which it will carry out to completion. Reactive architectures are the opposite of deliberative architectures. Pure reactive systems do not have planning capabilities. There is a reliance on external stimuli in order for it to move the robot. For example, a reactive-based vacuuming robot will not move until it detects a dust cluster nearby, and it will not go searching for it in a proactive manner.

For all intents and purposes, deliberative architectures are similar to (if not the same as) top-down architectures. Likewise, reactive architectures are similar to bottom-up architectures. Discussion of these types of architectures occurred previously in Section 2.3.

2.5 Architectures in Robotic Soccer

The potential of robotic soccer and why it is being used to evaluate the proposed architecture of this thesis was described previously in Section 1.1. To understand what makes a useful, flexible, and intuitive architecture (especially for robotic soccer domain), it is advantageous to understand the types of architectures used in the past robotic soccer competitions. This section will highlight some of the unique architectures that were used.

A popular hybrid architecture is the belief-desire-intention (BDI) architecture [27] which is being used by several robotic soccer teams [31, 32, 33, 91, 97]. Belief refers to the facts that an agent holds to be true about themselves and its environment. Desire refers to the goals of an agent. Finally, intention refers to the steps the agent plans to take to reach its goals or desires. The belief and desires of a robot help drive its long term planning strategies. The intention and desires of a robot help with its ability to improvise. This is a popular architecture since it is very flexible and the concepts themselves are intuitive. However, many aspects are implicit (e.g. conflict resolution), and thus it can be made more intuitive.

Many teams in robotic soccer competitions have used hybrid approaches. The MuCows from the University of Melbourne at RoboCup 2000 used a software architecture based on a set of modules that uses a form of message passing they describe as publishing and subscribing to messages [52]. One key advantage to their system is that it allows for the design of scalable and distributed systems, thus making their design very flexible. However, since much information is made implicit, it is not very intuitive.

At the RoboCup 2001 competition, the RoGi team from the University of Girona, Spain, used a unique multi-agent system [66]. Their design of the agent is controlled by a decision making module that is made from two parts: the reactive decisions, and the deliberative decisions. The former part is based on self-perceptions of the agent. The latter part requires communication between agents to create complex and cooperative behaviours. This cooperation allows for agents to reinforce the decisions made by the reactive decisions component.

The IUT Flash team used an interesting technique, fuzzy logic [67], to help the transition from one state to another [90]. This is an interesting idea since the dynamics of soccer does often suggest that state changes are more gradual in nature and not rigid with minor exceptions such as the referee making a call. Nevertheless, state machine approaches are highly integrated, which means that one state depends on another and thus makes it more difficult for behaviour designers to modify behaviours. Thus, it is not as flexible as it could have been.

XABSL

For many teams, much of the crucial information about the agents' play is implicit in the code. One exception is the team from Humboldt-Universität von Berlin, Universität Bremen, Technische Universität Darmstadt, Universität Dortmund and Freie Universität Berlin in the Four Legged League [37, 69]. They developed an architecture for Sony Aibo robot dogs based on Extensible Agent Behavior Specification Language (XABSL), in order to help describe behaviours (see Figure 2.26 for a sample behaviour definition) [68, 69]. It is important to note that XABSL is not

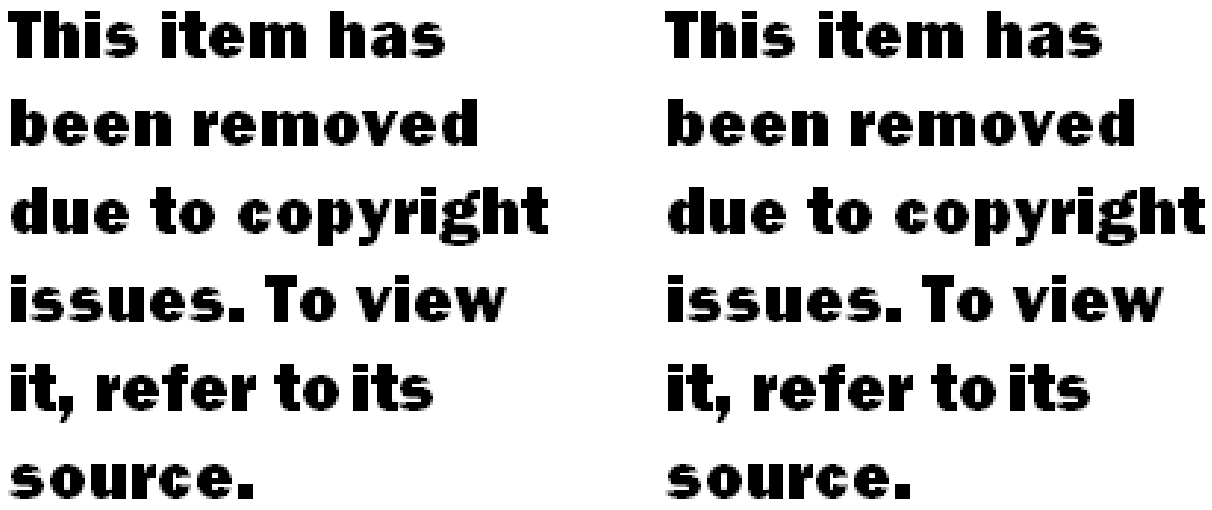


Figure 2.26: Sample XABSL striker definition, taken from [68].

an architecture itself. It is just one part (the behaviour system) of what makes an architecture.

XABSL is rooted in the use of *eXtensible Markup Language* (XML) [99], which means that information is made explicit to the developer. For example, it is easy to change the XABSL example above so that the strikers stay further behind the ball. Thus, using this explicitness approach is more intuitive than most of the previous approaches. However, there is some room for improvement to make it more flexible. For example, the code shown in Figure 2.26 does not allow for explicit flexibility in describing the destination point (i.e. no margin for freedom or variation).

2.6 Languages for Architecture design/implementation

To control the complexity and to simplify the design of architectures, researchers have also developed their own languages to help facilitate the creation of new architectures, as well as implementations based on existing architectures. XABSL, introduced previously, is one of the newer languages to have emerged from years of research. XABSL is intuitive since it allows developers to explicitly specify behaviours. It is also flexible as it has been implemented on two different robotic platforms. However, there are some shortcomings of XABSL, such as weak behavioural hierarchy support. The rest of this section describes some other languages that might be considered as alternatives to XABSL.

GRL

Horswill also describes a functional programming language called GRL (Generic Robot Language), which is used to program behaviour-based systems [51]. The GRL compiler allows programmers to write in a more modular manner, yet distills C code that is supposedly faster than hand-written code. Compared to raw LISP (short for LISt Processing, a language often associated with symbolic processing and high-level behaviour development), GRL was reported to be much easier to write and to debug. However, GRL can be made more flexible, since adding, modifying, and removing behaviours still requires much of the behaviour system to be recompiled and reloaded onto the robots.

ALFA

ALFA (short for A Language For Action) is another behaviour language for designing reactive control mechanisms for autonomous mobile robots [46]. It was first proposed by Gat in 1991. ALFA was designed to support bottom-up hierarchical architectures. Support for top-down, or other hybrid approaches are not well supported.

2.7 Summary of Related Work

From all the previous work described in this chapter, there is no one architecture that is flexible, adaptable, and intuitive. The previous approaches described here were all lacking in one area or another. The next chapter describes a new architecture, which uses several aspects briefly mentioned in this chapter (e.g. explicit representation), in a new combination that is intuitive, flexible, adaptable, and extensible.

Chapter 3

Design

The goal of this thesis work was to design a general purpose architecture. However, it is not necessary for the architecture to work for all imaginable robotic tasks. Further discussion for this reason is found in the next chapter. If an architecture is made more flexible, it is able to be adapted to nicely fit or better suit a larger variety of tasks. According to Pareto's principle [80], in anything, twenty percent is vital and eighty percent are trivial. Thus, it is more important to focus on the few vital tasks which are common among mobile robots. This is the principal upon which the design of the proposed architecture is based.

3.1 Requirements

In designing this architecture, the intuitiveness, flexibility, adaptability, and extensibility were deemed the most important. The survey of previous work in Chapter 2 shows that previous architectures fall short in one or more of these requirement areas.

To fulfill these requirements, the design will focus on the following aspects:

- *Separate and loosely-coupled sensor and actuator modules.* The mapping between sensor data and useful logical perception should be separated. Thus, the logical perception and the components that depends on these perceptions (e.g. the world model) can be implemented once, and then can be used by any behaviour in the architecture. Using a loosely-coupled methodology allows the architecture to fulfill the adaptability requirement. Further discussion about the implementation of this is found in Section 3.2.1).

- *Explicit world modelling with perceptual processing routines.* Using a world model helps the robot create plans for more goal-oriented tasks. The perceptual processing routines help extract useful perceptions from the raw sensor data. Having a world model helps fulfill the flexibility requirement. Further discussion of the implementation of this is found in Section 3.2.2).

- *Flexible sequencing support.* There needs to be multiple-types of sequencing, which help smooth out the overall behaviour and control of the robot. Also, the degree of sequencing needs to be controllable, in order to balance the reactivity of the robot to fit the task at hand. That is, the sequence length can be shortened or the entire sequencing feature turned off if the task requires more reactivity to the unexpected. In the implementation of the prototype for this proposed architecture, there are three levels of sequencing, with the higher levels performing more reasoning. This aspect helps fulfill the flexibility requirement. See Section 3.2.3 for more details.

- *Timing constraints support.* Quite often, tasks have timing constraints – for example, a minimum or maximum time a robot should spend on a task (see Section 3.2.4). In the proposed architecture, support for these constraints is specified within the explicit representation of the behaviour. The timing support fulfills the flexibility requirement, and the explicit representation of the timing constraints fulfills the intuitiveness requirement.

- *Flexible behaviour selection mechanism.* Behaviour selection refers to understanding what the robot should do in a particular scenario. As one can imagine, this is a difficult task itself. Quite often, behaviours are added or removed for various reasons (e.g. more precise localization due to additional sensors, or the environment has changed). The developer needs to add and remove behaviours with as little effort as possible in order for the system to be as flexible as possible. Further discussion regarding how this can be achieved is found in Section 3.2.5.

- *Explicit behaviour representation.* Making representation of behaviours explicit makes an architecture much more intuitive for developers. An explicit representation of the intent of the behaviour allows the developer to more easily debug and fix errors that otherwise would have gone unnoticed. Further implementation of this explicit representation is discussed in Section 3.2.6).

3.2 Design Overview

The new architecture presented here, *Archangel* (see Figure 3.1 for a general overview, and Figure 3.2 for more specifics), is a behaviour-based hybrid architecture that uses an explicit behaviour representation based on XML. Using a behaviour-based approach with behaviour abstraction, Archangel allows the developer to create simple behaviours, and more complex behaviours derived from these simple behaviours. As a hybrid approach, this architecture encompasses a deliberative aspect which allows the robot to create plans, and a reactive aspect to respond to unexpected scenarios.

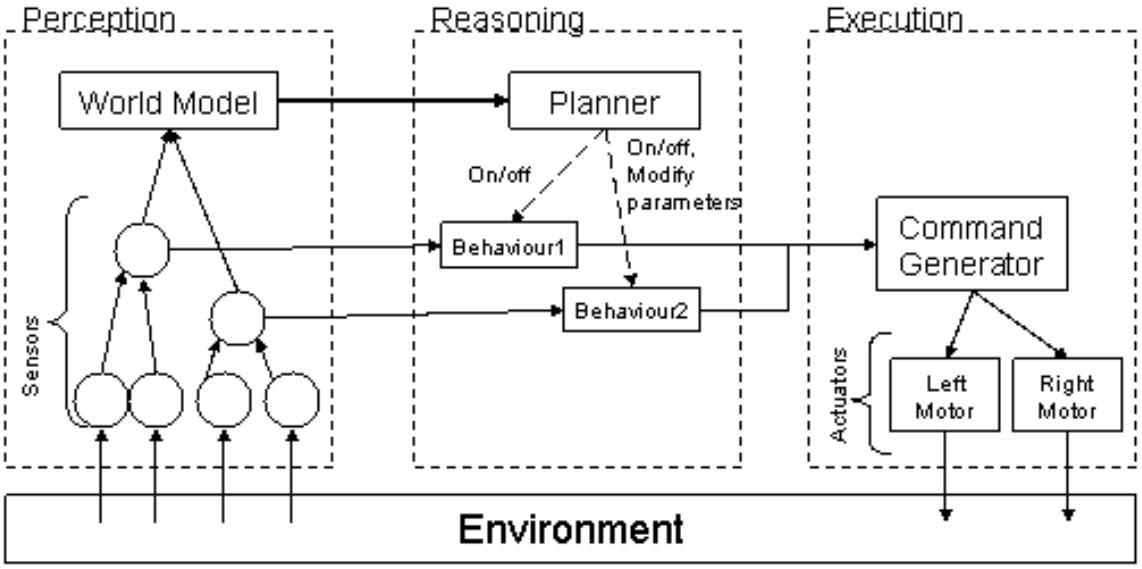


Figure 3.1: High-Level Overview Proposed Architecture.

Quite often in traditional systems, the trigger mechanism to execute a behaviour is too rigid and difficult to maintain. For example, consider maintaining and extending the behaviours in a large and complex decision tree. It would be difficult to know

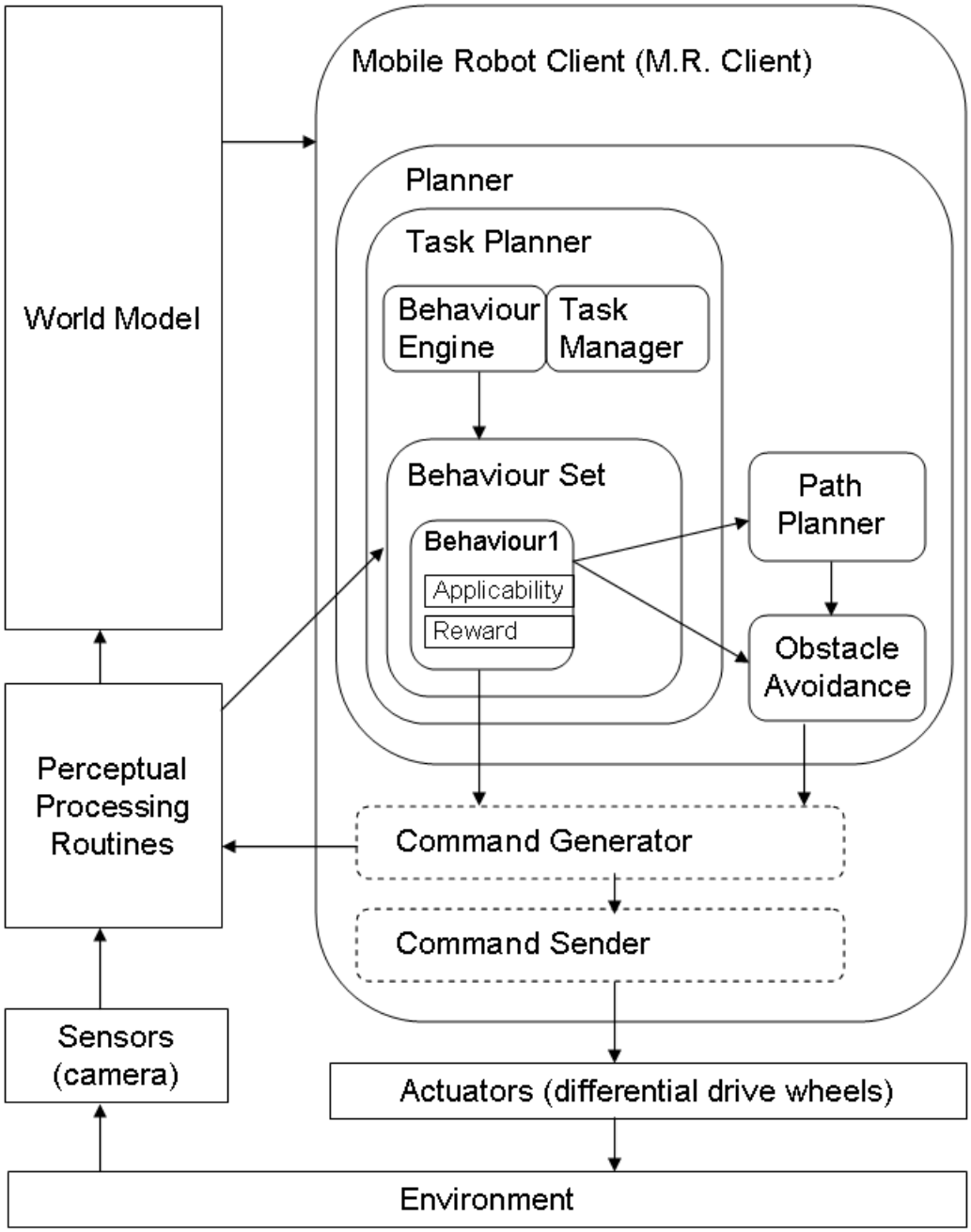


Figure 3.2: Overall Architecture Diagram.

exactly where to add new behaviours, and the impact to the system by removing a behaviour. Archangel deals with this issue by allowing the behaviour to specify how

applicable it is to a particular scenario. With Archangel, the developer can design the actions for the behaviour, and easily integrate the behaviour into an existing set of behaviours. See Chapter 4 for some examples.

3.2.1 Sensor and Actuator modules

The sensor and actuator modules were made separate from the rest of the system – an approach closer to Albus' RCS architecture [8] than Soldo's SSS architecture [88]. This is to increase the flexibility of the overall architecture and allow the architecture to work with different hardware and robots. For example, sensors and actuators can be swapped in and out with limited impact on the overall A.I. control system. If the sensors were tightly coupled with the behaviour system, this would lead to problems where modifications to the sensors and actuators would lead to large changes in the behaviour system. For example, instead of a global vision camera system, the designer added a local vision system and shaft encoders, then significant changes to the behaviour system would be required, such as modifications to all the behaviours. Using a loosely coupled design is more flexible. Also, this is more useful in designing and testing behaviours, since the behaviour designer will not have to focus on low level implementation aspects.

3.2.2 World Model

The world model is a globally accessible construct. It provides information about the environment and the objects contained within it (e.g. the robots, the ball, and obstacles). Some static or constant information it provides includes the size and

structure of the known environment (e.g. in the robotic soccer domain, this consists of the field dimensions and the positions of the goals). All the objects in the world model (with the exception of the walls) are considered mobile objects (whether they are self-powered or whether they require additional assistance is not considered). These mobile objects' positions and orientations are updated by the sensor modules (e.g. a global vision camera setup) and associated perceptual processing routines. The information the world model contains about each object consists of the type of object (e.g. robot, ball, obstacle, or unknown), the two dimensional (x, y) coordinates of the object, the global orientation, and the directional velocity (represented by x, y components). The coordinates represent the center of the object. The actual size of the object is determined by the type of object: robots are assumed to have a bounding radius of ninety millimeters, obstacles are assumed to have a bounding radius of one hundred millimeters, and the ball is assumed to have a radius of forty millimeters. This information is essentially read-only. Behaviours and other components (e.g. the low level controller) cannot make changes to the world model directly. The exception to this are the sensor modules, which are able to write information to update the world model. This information flow is shown in Figure 3.3. Information from the sensors will trigger an event in the *perceptual processing routines*. The perceptual processing routines take information from sensors and the controller, and update the world information. For example, a camera sensor module detects motion, which triggers a perceptual processing routine that filters the motion image to extract the moving object(s).

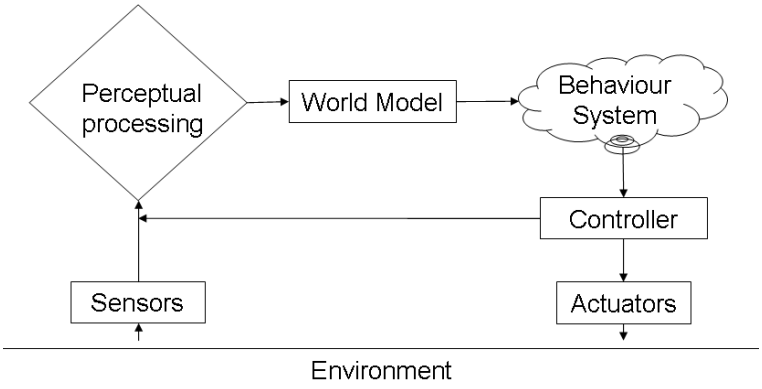


Figure 3.3: Information flow in Archangel.

It is important to note that the world model itself will not invoke or update the behaviour system. The behaviour system accesses the world model to extract the information that is needed, while the world model is a passive construct similar to the blackboard in blackboard-type architectures [30, 62]. It helps serve as a communication medium between the raw sensor data provided by the sensor layer and the useful logical information needed by much of the behaviour system.

3.2.3 Sequencing

The problem with sequencing can be broken into three separate levels: task sequencing, action sequencing and actuator command sequencing (see Fig 3.4). Sequencing further up the hierarchy requires more reasoning, whereas further down the hierarchy requires more low level knowledge of the system. It can be said that lower levels have finer granularity, since the sequenced components are smaller (i.e. performs less work and requires less effort to modify and debug). For this reason, lower level sequencing queues are often longer than the higher level sequencing.

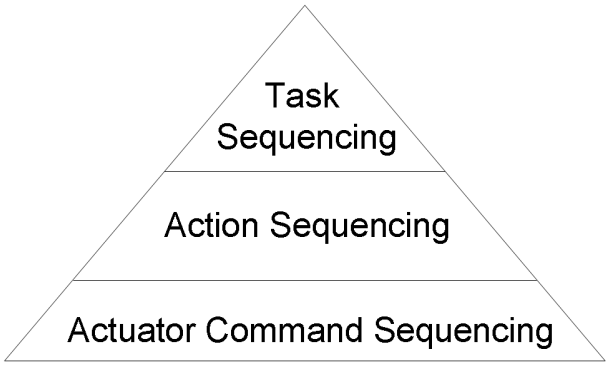


Figure 3.4: Sequencing Hierarchy.

For most systems, there is an overall "task" or purpose for the system. However, quite often there are subtasks involved in achieving this overall task. Task sequencing refers to ordering these subtasks in an appropriate sequence. In the Archangel architecture, part of this functionality is given to the behaviour system. The behaviour designer can specify task ordering in a behaviour. For example, there could be several states in a given behaviour, and the states can be ordered with conditions to help transfer from one state to another. The Archangel architecture also provides separate mechanisms to store previous states and behavioural tasks: a *history* of the internal state of the robot, if you will. This is useful for the reasoning system, and allows the behaviour designer to add additional control configurations. For example, using the history, the system can easily evaluate its progress and switch to a different behaviour if there is a lack of progress.

Action sequencing has finer granularity than task sequencing. The actions the robot should take to complete the task specified by the programmer are more of a matter of efficiency rather than some other type of requirements satisfaction. An example could be a set of waypoints to describe the most efficient path for the robot

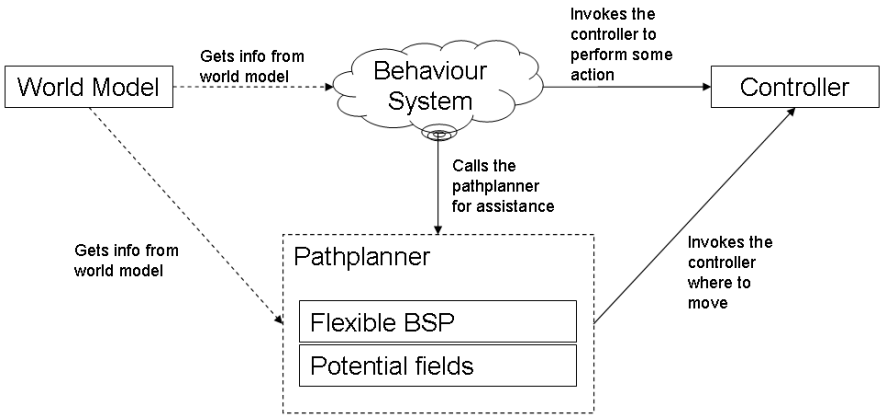


Figure 3.5: World Model - Behaviour System - Path Planner.

to travel to get from point A to point B. The path planner is the component that discovers this path (see Figure 3.5). It works as an *on-demand* component. The behaviour system will request the the best path given the source (the robot's current position) and the destination. The path planner can use any algorithm it wishes to find this path (e.g. Quadtree decomposition [78], Flexible Binary Space Partitioning [22], Potential fields [18, 58, 64], or other alternatives). The waypoints on the path are stored as a sequence, which can be later used to guide the robot. This is helpful if the environment is not expected to change rapidly.

Actuator command sequencing has the finest granularity among the three types of sequencing in the architecture. The purpose of this type of sequencing is mainly performance based. The advantages of using command sequencing are:

- less processing of complex routines (e.g. task and path planning) leading to more processing time for motor commands, which leads to faster motion;
- smoother and more natural turns;

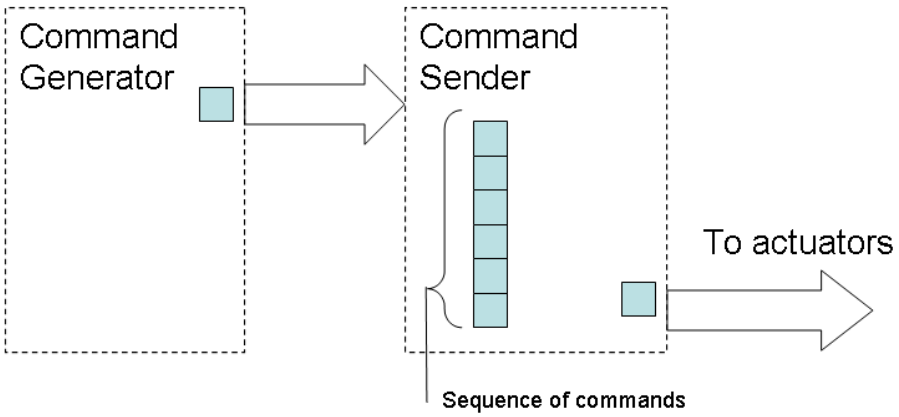


Figure 3.6: Information flow in Archangel.

- a reduction in the required processing power which can be used for other functions.

However, the disadvantage of having a long sequence is that reactivity is reduced. For example, a robot will continue a kick even though the ball has moved. As another example, a robot will continue to attempt to open a door even though the door has just been opened.

For actuator command sequencing, there are two components that deal with this issue: the command generator and the command sender (see Fig 3.6). The command generator takes waypoints or high-level motion commands (e.g. "kick") to produce low-level commands to the command sender. Designing this as a loosely coupled module in the architecture adds flexibility to change actuators. That is, changing actuators means a change in only how the higher level commands (e.g. the waypoints) maps to the low level actuators (e.g. how hard or how long to turn a wheel).

The command sender receives those commands and places them into a queue to be executed. The command sender attempts to send commands to the actuators at

regular intervals. However, some commands take longer than others to execute. Thus, the command sender will block the transmission of subsequent commands until the current one has completed execution. Currently, the mapping between the command sender and the actuators are fixed in the implementation phase (e.g. compile-time). Future work can be done in determining the benefits of making this aspect more reconfigurable.

3.2.4 Timing Constraints

There are certain tasks in any domain that have timing constraints. For example, a soccer-playing robot may have to wait ten seconds before it is allowed to steal the ball from the other team on a free kick. Alternatively, a floor-cleaning robot should spend at most thirty minutes cleaning before heading to its charging station if it is known the battery lasts only thirty-five minutes. In such scenarios, it is useful to set timing constraints on some behaviours. The Archangel architecture allows these constraints to be set within the behaviours themselves, giving maximum flexibility to the behaviour designer. Section 3.2.6 includes a detailed description.

3.2.5 MRClient Agents

Software agents called mobile robot clients (MRClients) are used to control each individual robot (see Figure 3.7). Each MRClient has its own planner, which consists of the behaviour engine (BE), task manager (TM), and a collection of behavioural schemes in an explicit representation as described in Section 3.2.6. It also has links

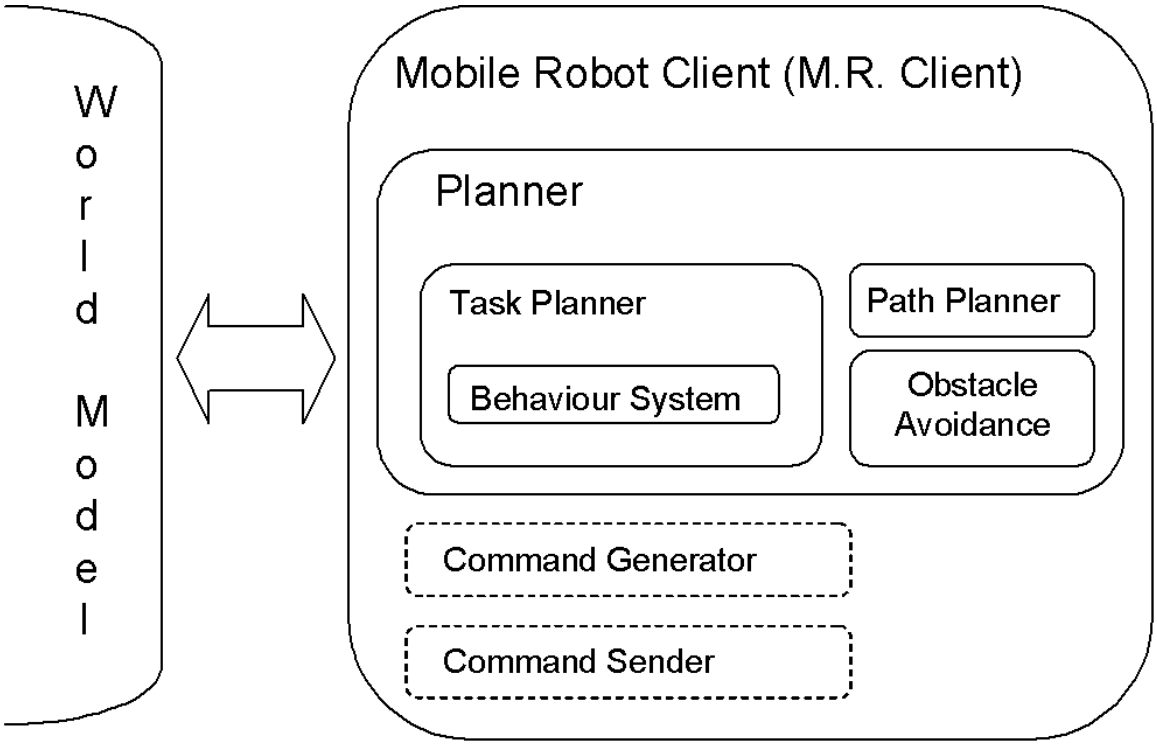


Figure 3.7: Mobile Robot Client.

to a command generator and a command sender. (The command generator and command sender can be local or global to the MRClient).

During the initialization phase at runtime, the behaviour engine reads in XML behaviours, parses them, and stores them internally as a collection of XML Document Object Model (DOM) objects to be interpreted. This provides a very flexible design (almost at the level of interpreted languages). With regards to the robots used for the evaluation (Chapter 4), this approach is very suitable. It is flexible in that it is easy to add, remove, and modify behaviours without large changes to other parts of the program, and without the need to recompile the program. Also, this higher-level of abstraction allows the behaviors to be validated independently regardless of whether

it was used in interpreted approach or a compiled approach. However, the interpreted approach is less suited for embedded systems which have limited resources to run an interpreter.

With regards to the robots used for the evaluation (Chapter 4), another possibility is the compiled approach where XML is used to define the behaviour and also embed "the how" or the specific implementation of the behaviour, which is then converted into a C++ class to be compiled with the program. This alternative suffers from a few disadvantages:

1. It is more difficult to add behaviours. The XML behaviour definitions would need to be converted to C++ code. The task planner must also be made aware of these new behaviours. Depending on the implementation, the designer also needs to be aware of how the new behaviour(s) will interact with the existing behaviours.

2. It is more difficult to modify existing behaviours. Small modifications such as moving a setup point behind a ball will require the XML behaviour definition to be converted to C++ again, and the program will need to be recompiled.

3. It is more difficult to remove existing behaviours. Modifications to the task planner are needed to remove behaviours from those available.

When the MRClient is invoked to move, either from an external stimulus such as an update to the world model, or from an internal timer, the MRClient will run its *execute function*, which is the main processing loop of the MRClient. The execute function will access the planner to decide on the correct course of action to take. The

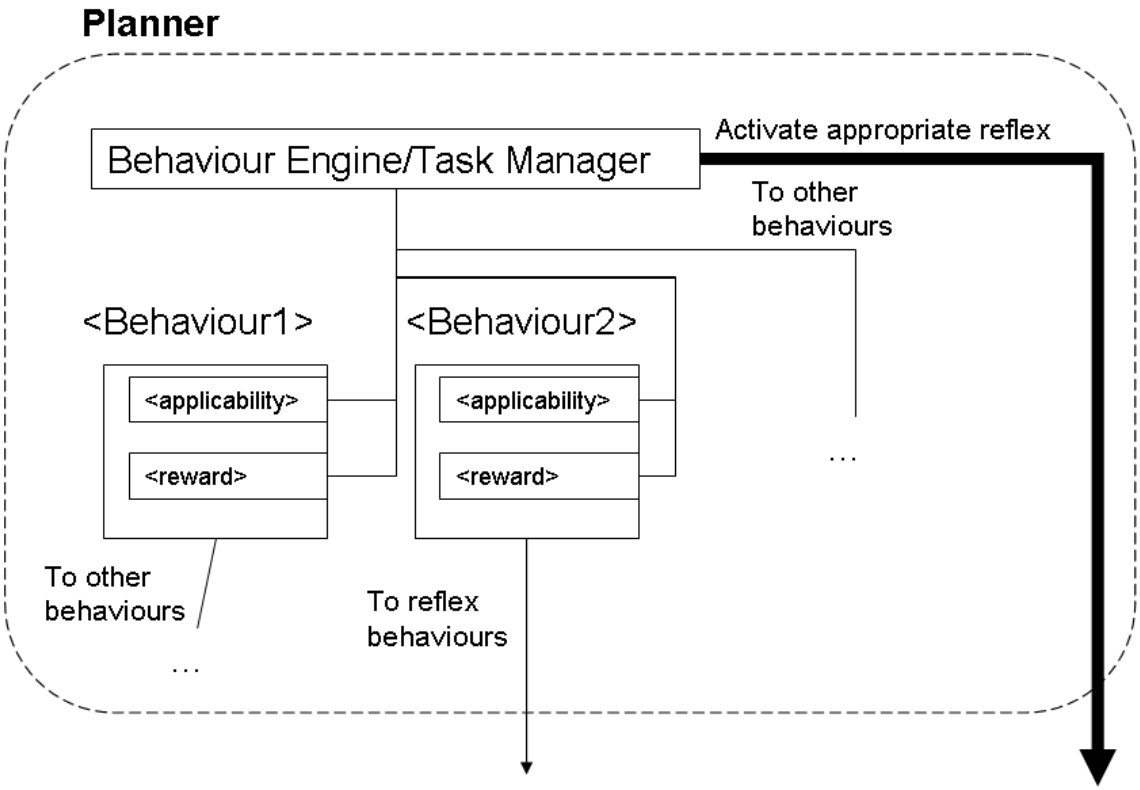


Figure 3.8: Planner: Behaviour Engine & Task Planner.

planner (see Figure 3.8) uses the behaviour engine to access the XML behaviours and then uses the task manager to decide which behaviour to activate.

Complex behaviours are represented as behaviour trees using abstraction. This is one way in which Archangel encompasses a deliberative approach. The reasoning behind using behaviour trees and abstraction is to create a system where it would be easy to add, remove, or modify complex behaviours. This is extremely important in situations where behaviours are to be reused. For example, consider a *GoHome* behaviour, which is a complex series of procedures a robot must do to head home (e.g. go to home area, signal security system, open the door, etc.). This can be created once, and then to include it as part of another behaviour can be as simple as adding

one extra line to that other behaviour. Removing the *GoHome* behaviour from the robot's set of behaviours can also be as simple as removing one line. Modifying this *GoHome* behaviour can also be done without severely impacting the overall behaviour system.

In addition, a complex behaviour can involve several internal states (e.g. a finite state automata – FSA). Rather than a tree-like structure with a somewhat linear path, these states can follow a complex graph-like structure with possibly cyclical paths. This is another way in which Archangel encompasses a deliberative approach.

Archangel uses a reactive component, the task manager, which uses competition as a means of behaviour selection [18, 65, 71, 72]. Behaviours will compete for the attention of the task manager (see Figure 3.9). This strategy allows for a reactive approach based on deliberative knowledge. When the task manager needs to decide on which behaviour to activate, it will query each behaviour about its *applicability* and its *reward*. The *applicability* of a behaviour refers to how applicable the behaviour is to the current world situation, while the *reward* refers to a reward value for completing that particular behaviour. All behaviors in the system support two functions in order to support this: *calc_applicability(state)* and *calc_reward(state)*.

The function *calc_applicability(state)* returns how close the current world state matches the assumptions of the behavior. For example, the behaviour *shoot-goal* returns a high applicability if the robot and the ball are lined up and close to the opponent's goal. The function *calc_reward(state)* returns the expected reward should the behavior succeed. For example, the *score-goal* returns a reward of 0.9 whereas *block-opponent* returns 0.3. The *calc_applicability(state)* and *calc_reward(state)* func-

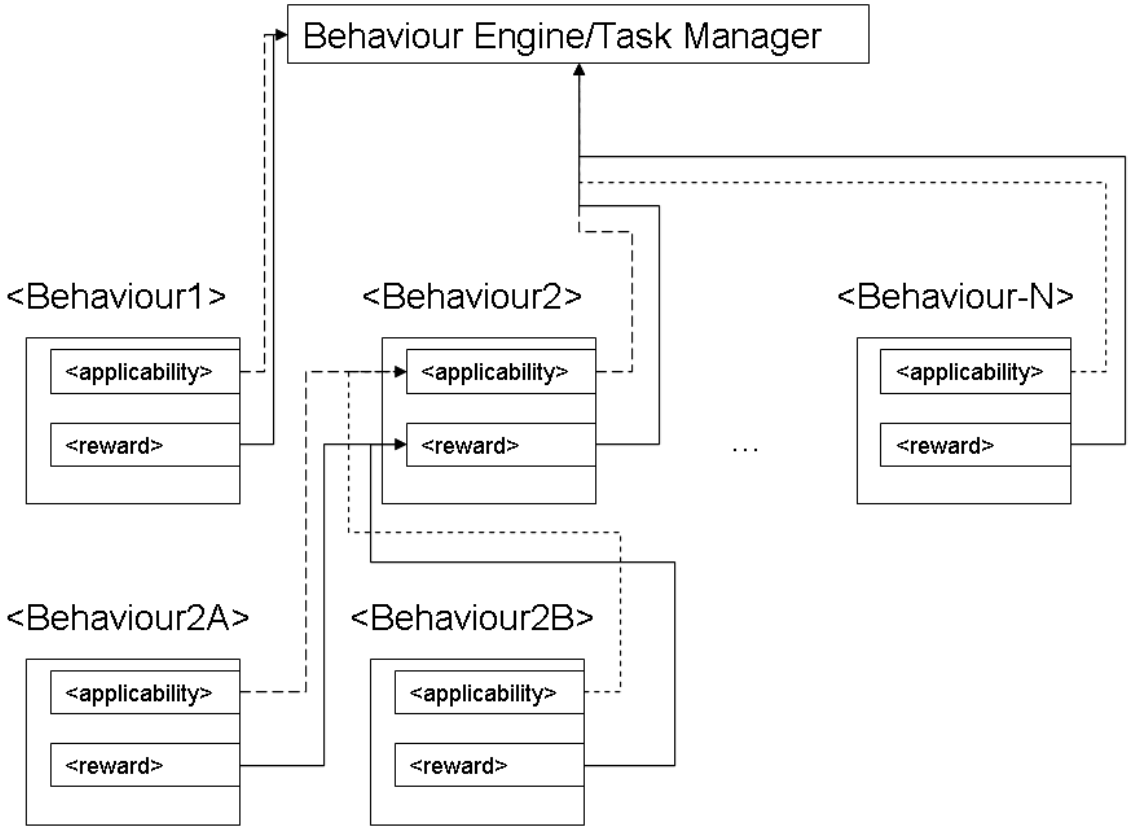


Figure 3.9: Competition in Archangel.

tions are limited in the amount of processing that they are allowed to do. These functions are expected to be approximations of the assumptions and effects of a behaviour, and are correspondingly expected to return very quickly. For example, to determine whether a given behaviour's assumptions are precisely met might require the behaviour to plan a complete path to the goal destination, which is computationally expensive. Instead, the *calc_applicability(state)* function could estimate this applicability by the distance of the robot to the goal position.

Given the applicability and the reward, the task manager will compute an *activation* value for that behaviour. The behaviour that has the largest activation value

will be the behaviour chosen by the task manager. The activation value can be described as a combination of the behaviour's applicability value and the reward value. This is a loose definition of an activation value to give developers some flexibility in how they wish to design this combination. However, a simple weighted sum has been implemented and shown to work well in the prototype program described in Chapter 4. Equation 3.1 describes the simple equation with A, and B being weights for the applicability and reward value respectively.

$$activationValue = A * applicabilityValue + B * rewardValue \tag{3.1}$$

Since behaviours can be specified from another set of behaviours, the behaviour engine and the task manager will work in a recursive manner. As shown in Figure 3.9, *behaviour2* is comprised of *behaviour2A* and *behaviour2B*. The applicability of *behaviour2* is a combination of the two sub-behaviours (e.g. max(*behaviour2A*, *behaviour2B*) – the maximum value between the *behaviour2A* and *behaviour2B*). A similar technique can be employed for reward components. Using this hierarchical abstraction control and action selection is a strong feature of the Archangel architecture. This approach is similar to that used by Decugis and Ferber [39]. The difference between Decugis and Ferber's architecture and Archangel is in the specification of the behaviour themselves. Archangel's behaviors are more intuitive and flexible, since relevant and important information is made explicit. This behaviour selection is also similar to that of the potential fields based architecture proposed by Laue and Röfer, however it differs from their approach since developers need not be familiar with the concepts of potential fields to design behaviours. Also, their behaviour selection

model of using potential fields works well for representing the majority of navigation behaviours, however it makes less sense in other scenarios. For example, it is harder to represent the activation criteria of a behaviour that calls for the robot to spin on the spot for a given length of time (e.g. a spin-kick for some soccer playing robots).

In Archangel, the chosen behaviour from the behaviour selection process (the one with the largest activation value) is executed until the next time step. With the use of competition as action selection, there can be situations where the robot will oscillate between two behaviours or goals, as Tyrrell showed in [94]. Tyrrell's experiments had the robot go to two separate locations, one for "eating" and another for "drinking". However, the control system constantly changed the predominant behaviour/goal and it never reached either location. To avoid the common problem of oscillating between behaviours, the task manager in Archangel enforces a minimum threshold for changing from one behaviour to another. Thus, the robot will likely continue on the path it initially chose. In cases such the one presented in Tyrrell's experiment, the applicability values in Archangel can be functions of the distance to the possible subgoal (eating location or drinking location). The robot will then likely choose to go to the closer subgoal. If both were equally close, the architecture's mechanism for this conflict resolution is always to choose the newer behaviour (the one with the latest information).

For future research, learning can be incorporated into the behaviours and task planner. Behaviours could be given the ability to adjust the value of the *calc_applicability(state)* and *calc_reward(state)* depending on the success of the behaviour. Other researchers are investigating methods for learning behaviour selection [41, 48, 77, 96].

Obstacle avoidance is an important feature often required of a robot in many domains. In the Archangel architecture, the obstacle avoidance behaviour is recognized as a separate component because of its importance. In the prototype Archangel program, compared to all the other behaviours, which were fully implemented in XML, a low-level obstacle avoidance behaviour (based on potential fields [58]) was implemented in C++. This was because the model of the potential fields itself rarely needs to be changed. However, all the parameters for this model are configurable in XML. For example, the force exerted by the ball by default is assumed to be repelling; however each XML behaviour definition can redefine this parameter to suit its needs. The "walls" in the world model will also exert a repelling force to help confine the robot within a specific area. Additional configurations can be added in the future. This potential fields behaviour also aids as a mechanism for conflict resolution. For example, in Tyrell's experiments previously mentioned, the robot will choose to go to the closest objective.

The next section will further describe the XML behaviour language used to describe individual behaviours.

3.2.6 Explicit Representation

The goal to make information explicit must be carefully balanced with the need for an intuitive architecture. Obviously, the most extensible architecture is a description of a C++ program in XML. However, the resulting system would be even more difficult to understand and modify than the original C++ code. Therefore, the type and amount of information that is expressed explicitly must be carefully controlled.

Explicit representation of behaviours is important for the architecture to be intuitive. XML was chosen as a basis for representation because it provides a foundation for developers to represent behaviours explicitly. Also, XML is flexible in its expressiveness by allowing the developer to freely describe a behaviour. Many systems are increasing their XML-support for analogous reasons and its use here allows this architecture to be implemented on many platforms and systems. Using XML also means that a variety of existing tools can be used to assist the programmer in writing the XML behaviour definitions. XML tools are constantly being improved upon such that they are almost to the point of having *WYSIWYG* (*What-You-See-Is-What-You-Get*) functionality, similar to the tools for HTML coding. This all leads to more intuitive tools for the future. Ultimately, the use of XML can potentially replace how programmers approach high-level specification design and implementation for robotic systems (i.e. replace some symbolic programming components).

The rest of this section will describe the XML foundation for explicit representing and describing behaviours. There is a considerable number of ways to represent behaviours, and the following representation is just one set of possibilities. It is important to note that this thesis is not primarily about using XML in the architecture – rather it is about the architecture design (and the specific implementation developed for robotic control). The set of XML tags described here was kept minimal so not to confuse/overwhelm developers with the numerous possibilities and to stay intuitive. An as-needed approach was used to develop this minimal set (i.e. a XML tag was created when the existing set of tags was insufficient for the task). However, this minimal set is sufficient to cover all the cases in the next chapter and thus the majority

of general situations that arise, which those cases are supposed to represent. The complete sample Document Type Definition (DTD), the document used for syntax validation, can be found in Appendix A.

The root element (top container entity) is the <behaviour> element. The <behaviour> element must have a unique name to help distinguish it from other behaviours. A behaviour can also have a minimum execution time associated with it. This allows a behaviour to let the task manager know how much time it will take, so the task manager will not switch to another behaviour during this time. This is a necessary characteristic in order to avoid switching between behaviours too often, which could result in behaviour oscillation with no progress being made. This relates back to the timing constraints support mentioned earlier in this chapter.

The components of a behaviour are: <init>, <draw_env>, <behaviour_list>, <reward>, <applicability> and <execute> (see Figure 3.10 – a sample high-level behaviour that contains sub-behaviours).

Behaviour components

The optional <init> section describes the variables this behaviour wishes to initialize. For example, the behaviour in Figure 3.10 initializes the <target_list>, which is a list of target positions. The <target_list> element has two primary attributes: *src* (the source to initialize the target list from), and *ofType* (the type of objects from the source). Currently the only source allowed is "World::videoObjects", which is the collection of objects given to the input of the system architecture. However, additional sources can be added in the future. The second attribute selects the type of objects (i.e. a subset of the source) to be included in the <target_list>.

```
- <behaviour name="sampleBehaviour">
  - <init>
      <target_list ofType="obstacle" src="World::videoObjects"/>
    </init>
  - <draw_env>
      <!-- Home location -->
      <pen colour="red"/>
      <rect x="360" y="40" width="20" height="20"/>
    </draw_env>
  - <behaviour_list>
      <behaviour_ref name="chaseTarget"/>
      <behaviour_ref name="goHome"/>
      <behaviour_ref name="dance"/>
    </behaviour_list>
    <reward value="0.5"/>
  - <applicability>
    - <robot fartherThan="50">
        <reference_position reference="closestTarget"/>
        <add value="0.8"/>
      </robot>
    </applicability>
    <execute useBehaviourList="true"/>
  </behaviour>
```

Figure 3.10: Sample XML behaviour.

The <behaviour_list> element lists all the lower level behaviours that make up the current behaviour. This allows for behaviour abstraction. Usually these lower level behaviours are simpler behaviours but this architecture does not make this a requirement. The <behaviour_list> element contains multiple <behaviour_ref> elements, which are equivalent to *pointers* (for those familiar with C/C++) to other behaviours. The <behaviour_ref> element has a required attribute, *name*, which is the name of the sub-behaviour.

The <reward> section is intended to describe the reward the robot will receive when it successfully completes this behaviour. The behaviour designer can use this element to help design a reward function that can be used for behaviour learning applications. A final reward value for the behaviour can then be generated. The reward values in the Archangel prototype is from a range of 0 to 1, where 0 being no reward and 1 being the maximal reward.

The <applicability> section describes how applicable the behaviour is to a certain scenario. This is done by providing a list of conditions that needs to hold true in order the behaviour to be considered applicable. These conditions modify an applicability value from a range of 0 (not applicable at all) to 1 (very applicable). Each condition is described using a <condition> element. Alternatively, a fixed specific value can be used as well (e.g. <applicability value= "0.75" />).

The <condition> element allows for three possible types of child tags to help describe the scenario: <robot>, <ball>, <shot_on_goal>. The latter two are more applicable to the specific domain of a soccer playing robot, however, for other domains, additional tags can be added. The <condition> element also allows for a value to override whatever is the described condition by assigning a Boolean value to the attribute *met* (e.g. <condition met= "true" />).

The <robot> element specifies the state of the robot. Since the intent was for this to be used by *mobile* robots, this element is used to describe the robot's position. Similarly, the <ball> tag specifies the state of the ball.

The <execute> section describes the actions for the robot to take for this behaviour. For simple behaviours, the following simple actions were implemented for a mobile robot to take: <goto>, <turn>, and <kick>.

The <goto> element requires a position to which a robot can move. To describe a position, different specifications were implemented:

- <absolute_position> – describes an absolute position in global coordinates expressed as an x and y value.

- <reference_position> – describes positions that are well known in the world model, whether they are fixed positions (e.g. home locations) or relative positions (e.g. the ball).

- <relative_abs_focus_position> – is a modification to the previous specification in that it allows relative positioning to a reference point. This is done by providing a second focus point expressed using *x, y* coordinates. The relative position can be a point between these two points, or just a point aligned with the first two points. For example, the following XML code describes a position behind the ball, aiming at the point with coordinates (2700,800).

  ```
  <relative_ref_focus_position offsetPos=‘‘behind’’
  reference=‘‘World::ball’’ focusPoint.x=‘‘2700’’ focusPoint.y=‘‘800’’/>
  ```

- <relative_ref_focus_position> – is a further modification in that the focus point can be described using another focus point. For example, assuming the coordinates of (2700,800) represented the center of the opponent's net, the pre-

vious example can be rewritten as:

```
<relative_ref_focus_position offsetPos=‘‘behind’’
reference=‘‘World::ball’’ focusPoint=‘‘World::theirGoalCenter’’/>
```

This representation is easier to understand since significance of those coordinates is made more explicit.

In most cases, the robot will not need to be absolutely perfect in its positioning, and it is also difficult to ensure this. Due to noise in the sensors (e.g. vision system) and small positional errors in the world, some flexibility must be allowed with this respect. The <goto> tag has an attribute, called *within*, to allow for a margin of freedom (i.e. an offset). This is one feature that distinguishes this explicit representation from that of XABSL mentioned in the previous chapter. The only drawback to this offset representation is that it does not distinguish the direction of the offset. The latter two position elements provide additional support for this finer offset control. However, for most purposes, the first two specifications should be adequate. Additional elements can be defined in the future if needed.

The <turn> element is a very simple action in that it requires a point that the robot needs to face towards or away from. For example, the following XML code directs the robot to turn towards the ball.

```
<turn direction=‘‘towards’’>
    <reference_position reference=‘‘World::ball’’/>
</turn>
```

The <kick> element is another very simple action which instructs the mobile robot to move its lower appendages (if any). In the Archangel prototype, a kick

was defined as a quick motion forward, backward, to another specified direction. For example, the following XML code describes

```
<kick type=‘‘forward’’/>
```

Theoretically, using <goto> can result in the same action, however defining a <kick> element is much more explicit in describing the robot's intended action.

The <execute> section describes the actions for the robot to take to achieve the goals of the behaviour. It may also contain a FSA, using the <state> element. Each state in the FSA will contain one or more of the simple actions, a reference to the next (default) state (<next_state>), and the condition that has to be met in order for it to change to the next state (<condition>, a child of <next_state>). Note that the next state can be set to be recursive, which is useful to act as an end state. Each state can also have a minimum execution time associated with it to delay transition to another state to prevent certain problems such as oscillation between behaviour states.

In addition to the features described previously, each state may also contain a set of triggers to allow it to change to more than one state [18]. The usefulness of this is shown in the next chapter. Each trigger contains a reference to a state, along with the conditions that needs to be met in order for the transition to occur. However, this trigger set is optional (see Figure 3.11). This trigger set mechanism is also used for the child behaviours as well. This allows for a more reactive approach whenever possible.

The <draw_env> section is used for drawing, or physically displaying information and is intended primarily for debugging purposes and to let the developer know

```
<!-- trigger if reach end zone 1 -->
- <trigger_set>
  - <trigger_next_state targetName="raceLeg1End">
    - <condition>
      - <robot>
          <within_rect x="2520" y="260" width="220" height="1000"/>
        </robot>
      </condition>
    </trigger_next_state>
  </trigger_set>
```

Figure 3.11: Example XML trigger example.

which behaviour is executing. The <draw_env> tag has the following child elements: <pen>, <line>, and <rect>. Additional items can be included in the future. There can be multiple items and they can appear in any order required. The <pen> element cannot have any child nodes and has a required attribute: *colour* (which describes the pen colour). The values for colour are taken from the set of enumerated colours in the QT libraries [55] (which were used for the implementation). The <line> element does not have any child nodes, and has four required attributes: *x1, y1, x2,* and *y2*, which represent the coordinates for the line. The <rect> element does not have any child nodes, and has four required attributes: *x1, y1, width*, and *height*. The first two attributes represent the coordinates of the top left corner of the rectangle, and the width and height attributes describe the dimensions of the rectangle. Figure 3.12 shows how the drawing aspects is used to display information for the user in a racetrack task, which can be used as feedback in determining how well the robot is following a path and to show the target destination of the robot (see Section 4.1 and Section 4.3.1 for specific information relating to this task).

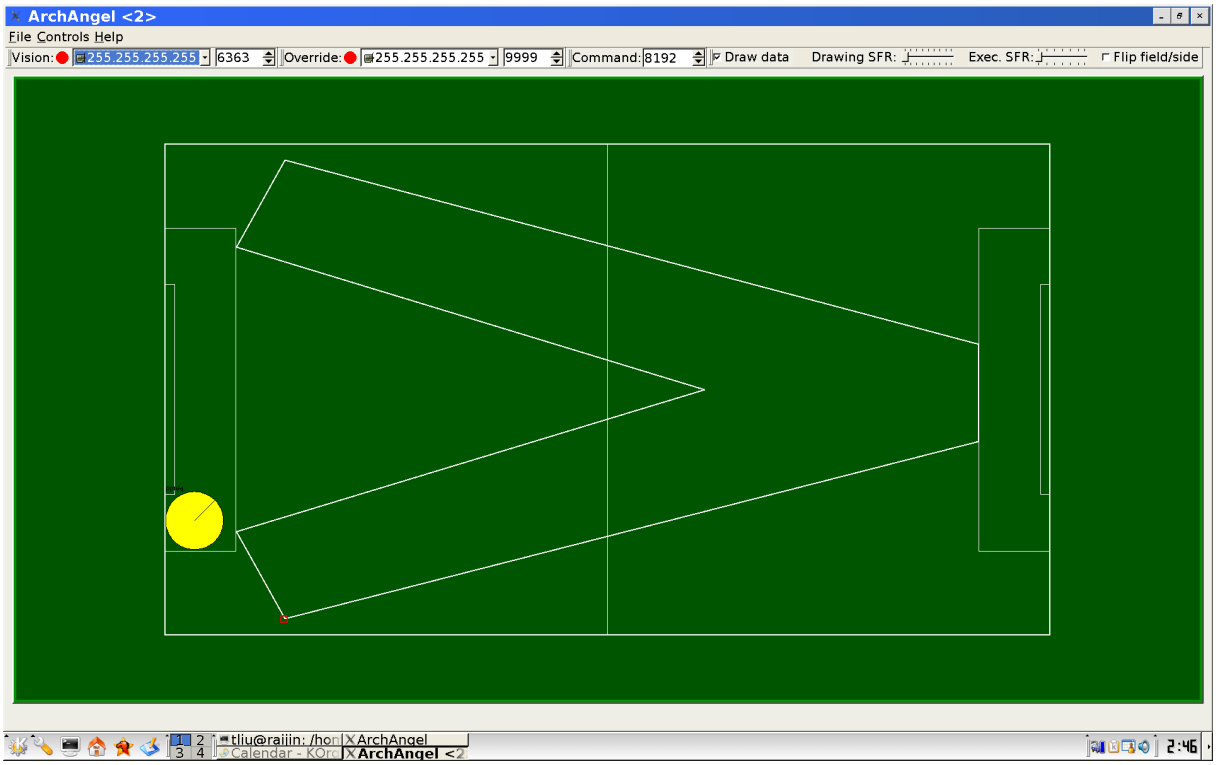


Figure 3.12: Screenshot of prototype program.

In summary, with the supported features and components described in this section, Archangel is designed to be an intuitive, flexible, and adaptable architecture. The next section will describe some implementation details and how Archangel was evaluated.

Chapter 4

Evaluation

This Chapter describes the methodology used for evaluating the Archangel architecture and the results of the evaluation. A prototype of the architecture was implemented and used to control several robots in a series of challenges, which represent common tasks performed by mobile robots. This development process was compared against similar programs created by the UM RoboBison team [10, 11, 12].

While some aspects of an architecture can be easily evaluated (e.g. average runtime of the control cycle), measuring the extendibility or flexibility of an architecture is difficult since these aspects are subjective and vary from user to user. Furthermore, concepts such as flexibility are subjective in themselves.

Arkin [15] stated that the major distinction between architectures is not a matter of computability, but rather efficiency, which argues that certain architectures are better suited for certain tasks. He further suggests that comparing architectures is similar to arguments made between programming languages (e.g. C++ vs. Java vs. Lisp). I argue that not only is the efficiency of the final system important, but also

the ease with which modifications and debugging can be performed. To extend Arkin's analogy, comparing architectures is similar to comparing how quickly different tasks can be solved with different programming languages. This suggests that comparing architectures itself is pointless unless one compares their performance on specific tasks. Recall from Chapter 1 and Chapter 3, Archangel was intended to be more of a general robotic control architecture rather than one that was focused on one particular domain. However, the prototype is grounded in robotic soccer to help with the evaluation.

To achieve meaningful quantitative results in an architectural evaluation requires large user studies, which is far beyond the scope of a Master's thesis. Therefore, anecdotal evidence, which was collected during the implementation of several challenge tasks, will be used to evaluate features of the architecture. The chosen challenges represent common tasks required of mobile robots. The series of challenges that will be used as benchmark problems for this new architecture are as follows: (a) path tracking on a racetrack, (b) path planning and navigation in a treasure hunt, (c) obstacle run, (d) shooting/goal scoring, and (e) ball passing. Some of these challenges have been used elsewhere in the robotics community [4, 6].

Section 4.1 will further elaborate on the challenges used to evaluate the architecture and why they are used. Section 4.2 describes additional relevant implementation details that relates to the evaluation of the robots. Section 4.3 describes the actual evaluation of the architecture, and provides the evidence mentioned previously. Finally, Section 4.4 will summarize the evaluation and display some of the results. Part of the results was based on past experiences with previous C++ programs developed

over the years. However, it is important to note that the results are misleading if one does not read or understand the reasoning behind the results, detailed in Section 4.3.

4.1 Tasks

The first challenge is the path tracking task, which demonstrates the behaviours involved in path following. Mobile robots are commonly expected to move along a predetermined path. This task will be grounded in a racetrack environment that involves the robot racing around a custom racetrack for a fixed number of laps (see Figure 4.1). The racetrack has multiple turns to make it more challenging for the robot, as this requires the robot to have better timing and control. This challenge is a time trial with only one robot on the track. Normally, the performance of the robot is measured by the total time it takes the robot to complete the course. Penalty times are assigned when a robot performs an illegal action (e.g. cutting a corner). To evaluate the architecture, this particular racetrack may change shape (i.e. the number or type of turns may change).

The Treasure Hunt (see Figure 4.2) is another timed challenge, which demonstrates path planning and navigation behaviours. This challenge involves a robot searching an area for all the target items (*treasures*) scattered randomly about a field. The robot will move to each target in turn. As an additional requirement, to signal that the robot thinks it has reached a target point, it must stay in the same spot on top of the target point for five seconds. This challenge tests the robot's effectiveness in finding a path to cover all the targets in the fastest time possible (compared to the time of the robot's previous trials and that of other robots). Normally, the performance measure

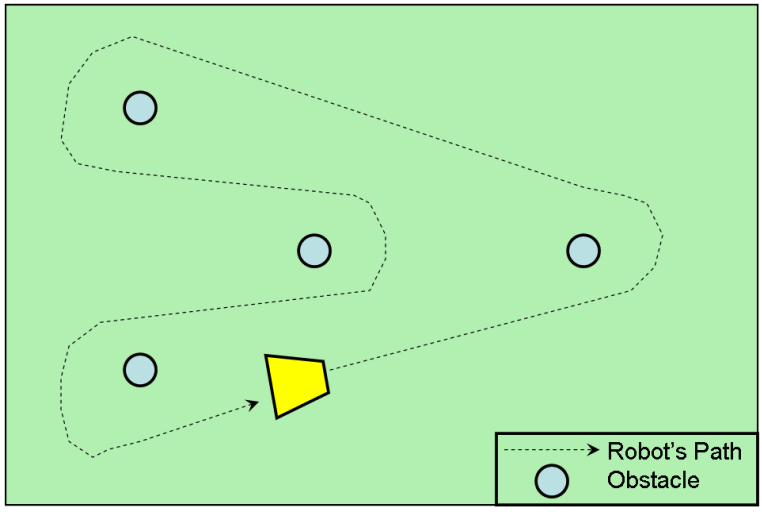


Figure 4.1: Racetrack.

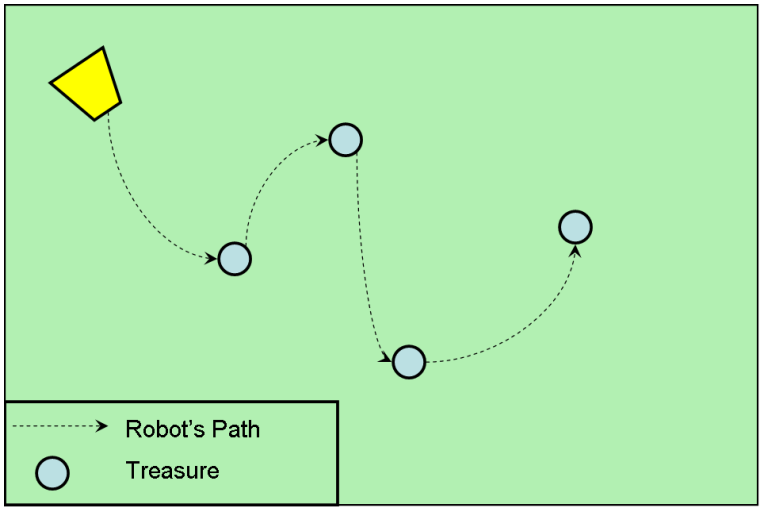


Figure 4.2: Treasure Hunt.

in evaluating the robot is the number of targets the robot can visit within the allotted time. To evaluate the architecture, the treasures will be randomly placed in different locations over several trials between passes. Other requirements and constraints can

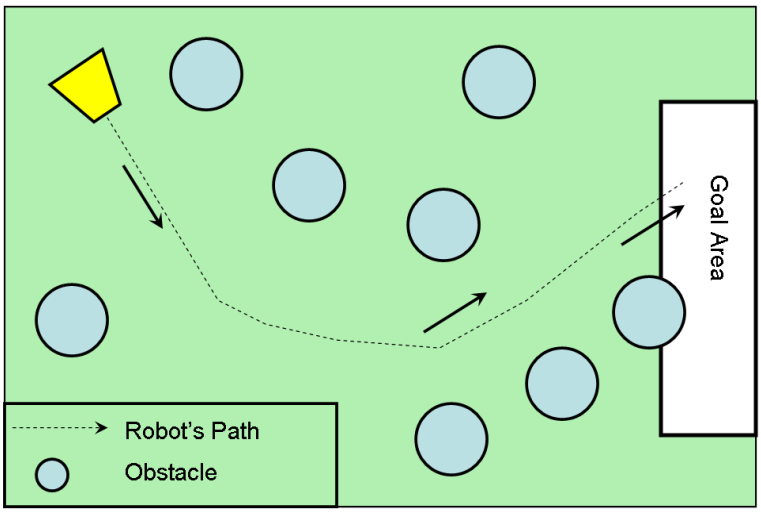


Figure 4.3: Obstacle Run.

be imposed in the future to see how easy it is to change the behaviours to adapt to new situations or cope with constraints.

The Obstacle Run (see Figure 4.3) is another timed challenge, involving the robot running from one end of the field to the other for a fixed number of times, while avoiding obstacles on the field. This challenge demonstrates obstacle avoidance behaviours. Normally, the performance measure in evaluating the robot is the time it takes the robot to complete the course. The robot is penalized for touching the obstacles by extra time being added to their trial time. To evaluate the architecture, the obstacles will be randomly placed around the field over several trials. For some trials, additional constraints such as a time delay (having the robot stay in the same spot for a certain amount of time) were imposed.

The Goal Scoring challenge (see Figure 4.4) has the robot attempting to kick the ball on an empty net, gathering as many goals as possible in a fixed time period. This challenge demonstrates goal-oriented (or task-directed) behaviours. This is similar to

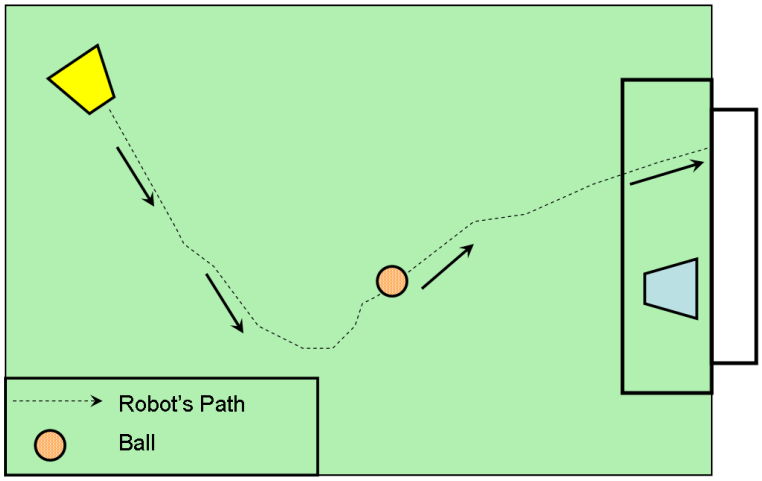


Figure 4.4: Goal Scoring Challenge.

the popular scavenging (search and gather) domain in A.I. The performance measure in evaluating the robot for this task is the number of goals scored within the time limit. To evaluate the architecture, additional constraints were imposed, such as a static goalkeeper. This challenge also involves some aspects from the obstacle run challenge, such as obstacle avoidance. The robot needs to avoid the ball in some scenarios, in order to reach a position behind the ball (so that the ball can be kicked). However, the ball needs to be considered an obstacle only part of the time – otherwise, the robot will never approach it for a kick. Using a static goalkeeper adds an additional element of complexity in that it will be a permanent obstacle.

The final challenge is the passing challenge. This challenge (see Figure 4.5) is more complex than the other challenges and is designed to demonstrate interaction between two robots. It involves two robots passing the ball between each other on a field divided into four quadrants. The field is first divided in halves, where there will be one robot on each half. The field-half each robot is on is further divided into halves

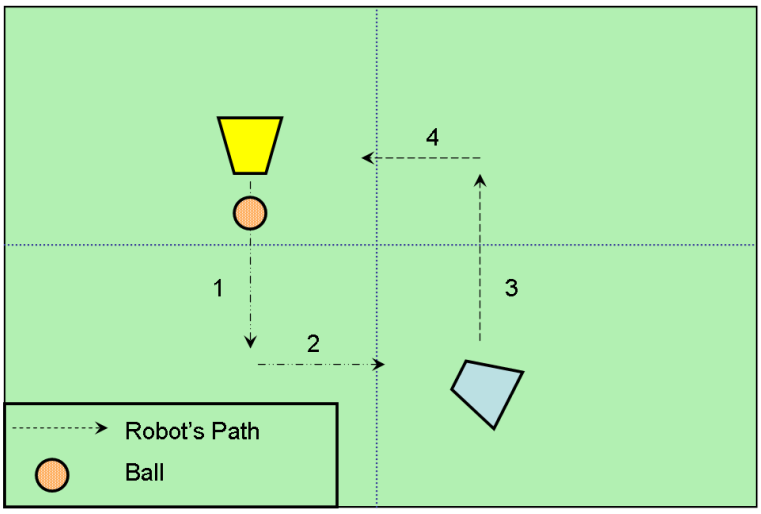


Figure 4.5: Passing Challenge.

again. When a robot receives a pass from the robot on the other half of the field, the receiving robot must dribble the ball to the other quadrant on its own half of the field before attempting to pass to the other robot. The robots attempt to make as many passes as they can in a fixed time period. The performance measure in evaluating the robots is the number of successful passes within the time limit. An unsuccessful pass refers to a scenario where this cycle is not completed. For example, if a robot kicks the ball off the field. Subtle changes to the passing techniques will also help evaluate the flexibility of the architecture.

4.2 Implementation details

This architecture has been implemented on two types of physical robots: 1) infrared (IR) controlled tanks, and 2) robots made from the Lego Mindstorm kits (see Figure 4.6). The IR tanks use a fixed infrared command protocol. For both these

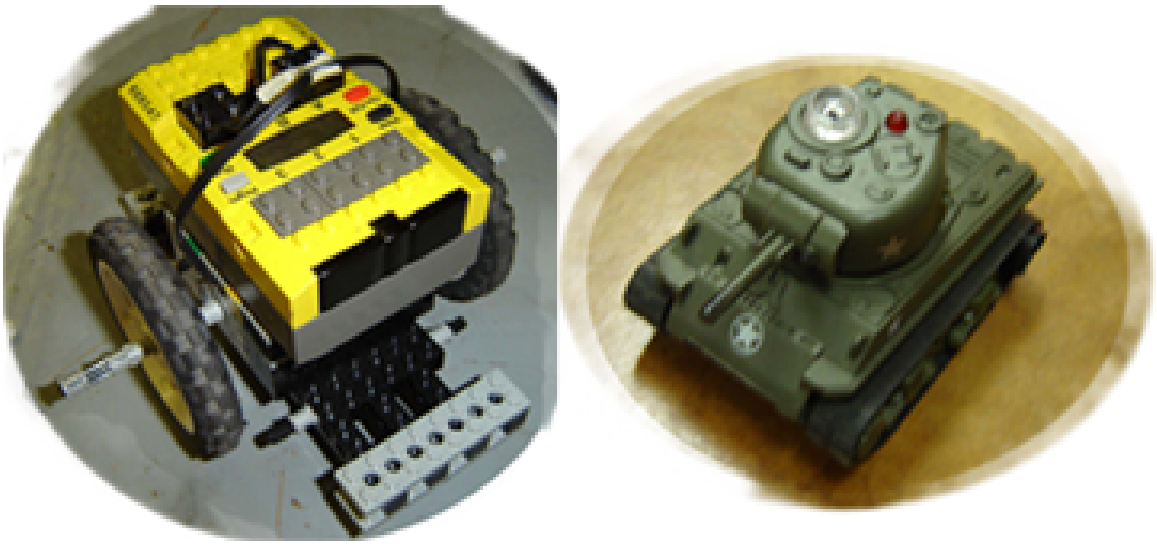


Figure 4.6: Lego Robot and Tank Robot.

robots, off-board C++ applications were written using the QT libraries from Trolltech [55] to demonstrate the architecture in practice. Because of the flexibility of the architecture, the most difficult modification was the change of the command generator: a Lego Mindstorm command generator was required for the Lego robots, and a separate command generator was required for the IR tanks. This was to control the timing to generate suitable velocities on the robots because of the physical differences between them (e.g. the tanks are a quarter of the size of the Lego Mindstorm robots). The other modification was a small change to scale the environments the robots will be tested in (due to the difference in size between the robots and the available of materials used to construct the fields). All behaviours were reused (with minor changes for efficiency – see Section 4.3.2 for an example) demonstrating the flexibility of the architecture.

A global vision camera was used for sensors. This global vision setup uses a low-grade camera mounted from an angle instead of being mounted directly overhead. This type of setup introduces some problems such as occlusion, to which the system must adapt. A separate vision processing system called Ergo Vision Server [45] (evolution of the Doraemon Vision Server [3, 24]) was used for simplicity, since it is a robust system that was especially designed for angle mounted camera systems. The vision server transmits filtered position data to the program controlling the robots via User Datagram Protocol (UDP) messages across a network at roughly twenty-four to thirty frames per second for real-time processing. UDP does not guarantee a successful transmission of any packet, but it is suitable for robotics research. Using Transmission Control Protocol (TCP), which uses retransmission as a mechanism to compensate for packet collision, would only serve to force obsolete information to be retransmitted. UDP, on the other hand, avoid these retransmissions, since that data may no longer be pertinent to the state of actual the world. However, due to imprecision/noise in the sensor system (and partially to the inherent unreliability of UDP), the Archangel program's sequencing was disabled because the hardware did not receive all the commands and therefore it was having the robot move off course. The effect was that the resulting control behaviours were more reactive and able to recover faster from errors.

All high-level behaviour definitions were designed in XML in order to meet the requirement for explicit representation (and ease of behaviour modification). This required a significant amount of work (e.g. implementing a module to parse, validate, and interpret the XML behaviour definitions). To simplify matters, the QT libraries

contain XML parsing modules. For this implementation, the QT Document Object Model (QDOM) module was used. This particular module reads and parses the XML file only once, and then creates an XML Document Object Model (DOM) Tree, which is stored in memory for later use. However, it does not have full XML validation capabilities. That is, inputs are expected to be valid XML files. Otherwise, an error is returned by the QDOM parsing module. It does not specify the specific error, or the location of the error, because of its limited validation capabilities. To deal with this, I validated input using another third party tool, xmllint [56]. Most of the real work was involved in designing the XML language specification to define the behaviours for mobile robots and the developing the interpreter to translate the behaviour specifications into observable actions. In addition, validated XML behaviour definitions, by themselves, do not ensure proper operation of the mobile robot clients. For example, it is impossible in XML to specify limits for numerical values. Thus, error and sanity checking needed to be enforced in the implementation itself.

4.3 Evaluation of the Challenges

This section provides some anecdotal evidence for evaluation. Previously developed control programs (e.g. the programs developed for UM RoboCup teams in 2003-2004 [10, 11, 12]) for these tasks were implemented in C++ only. However, the 2004 client program implemented some parameters in a separate text file. The Archangel program takes this initial concept further by making the entire behaviour components easily configurable in a high-level representation using XML. Evaluating these challenges involved comparing the Archangel prototype against the 2003 and 2004 C++

client programs. For the evaluation, I will play the role of the developer making the changes to the existing code, since it was impossible to get all the original developers of these programs to participate due to time constraints and other commitments. Also, because I was one of the original developers of each of these program, my familiarity with these programs will greatly impact these results. That is, it will take less time for me to make such changes compared to an untrained developer. However, since I am familiar with each of these programs, the results for each of the program should be affected equally. On a related note, because the original C++ programs were not designed to work with the IR tanks, the timing results and code changes for the original C++ program relate only to the Lego Mindstorm robots.

4.3.1 Racetrack Challenge

The first challenge, the racetrack, is a general path following control problem. In the original C++ programs, the path was represented by an array of *x, y* coordinates. For the new program, one XML behaviour definition was created, with several internal states (see Figure 4.7). Each state shows that a robot needs to go to a certain position and the conditions for moving to another position.

In Figure 4.8, the state is simply named *state1*. The action of this state is to move to a position in the world specified by absolute coordinates. The condition for moving to the next state is dependent upon the distance between the robot and the current target point (e.g. sixty millimeters).

One important specification of this state is that it specifies some graphical feedback is to be drawn; in this case, a red twenty pixel square at the current target point.

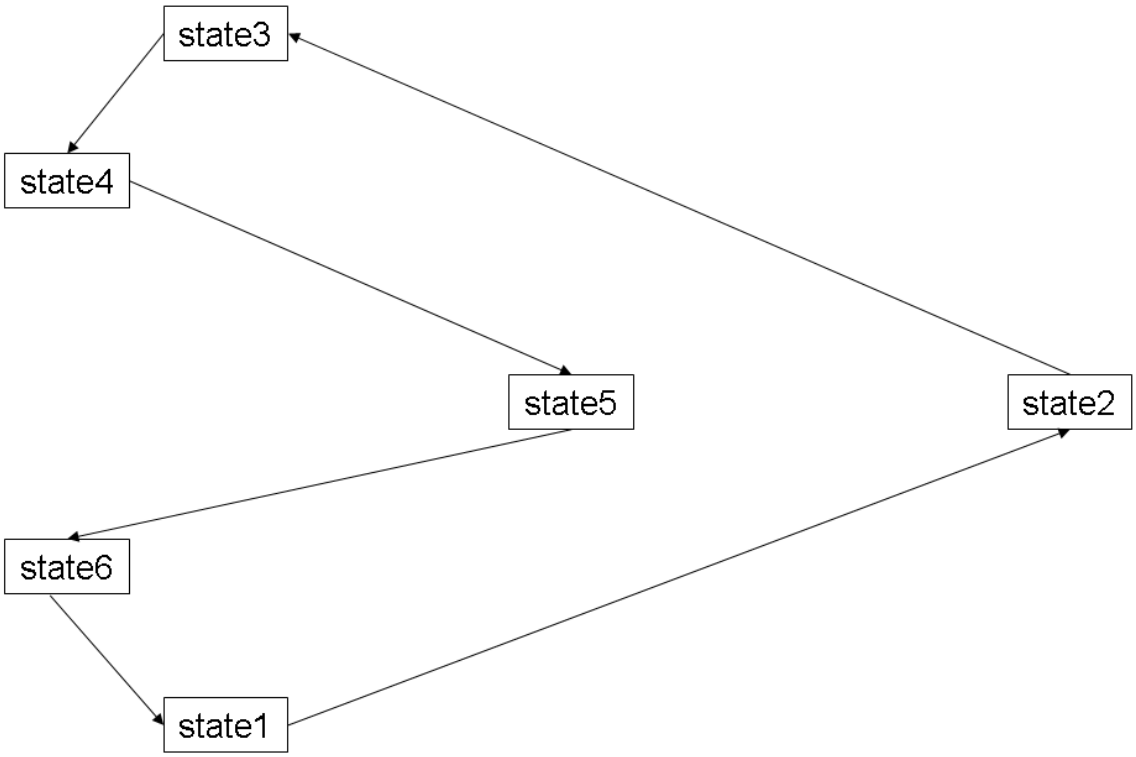


Figure 4.7: Initial Race FSA.

Graphical feedback is more meaningful and intuitive compared to textual feedback. In drawing a target point on the field, it also represents the relative distance to the robot, which is easily understood by humans. This is significantly simpler than reading tens to hundreds of lines of debugging output statements. Drawing this rectangle significantly simplified the debugging phase. Initially, the robot's movement were extremely random looking. But looking at this alone, it was difficult to know if it was the behaviour system, or another problem elsewhere (e.g. misidentification of objects due to noisy vision data, IR communication drop-outs, etc.). After the addition of this square, the behaviour system showed the square was at a constant location, which pointed to a failure of another component. In this case, the problem

```
- <state name="state1">
  - <draw_env>
      <pen colour="red"/>
      <rect x="360" y="40" width="20" height="20"/>
    </draw_env>
  - <goto>
      <absolute_position x="370" y="50"/>
    </goto>
  - <next_state name="state2">
    - <condition>
      - <robot within="60">
          <absolute_position x="370" y="50"/>
        </robot>
      </condition>
    </next_state>
</state>
```

Figure 4.8: Sample Race Track State.

was due to noisy vision data and was solved by re-calibrating the vision system. The simple addition of this square is estimated to have saved hours worth of debugging. The other states are similar to this one shown in Figure 4.8.

Developing code for this challenge in the Archangel program took roughly 4 days. However, this was because much of the behaviour engine's foundation still needed to be solidified (i.e. it was still being developed). Re-implementation of another path following task should take significantly less time now that this is completed. Alternatively, development of this task in the original C++ programs took roughly a week (given past performance of students of a Mobile Robotics Course taught at the University of Manitoba [1]) and did not include the development of an overall behaviour engine.

The code size for the behaviour itself was smaller for the original C++ program. Much of this was because the XML behaviour definition of the Archangel program included some repetition. However, it was somewhat difficult to debug this task in the original 2004 C++ program. Control information was distributed among multiple files, and part of the code was implicit in the design (i.e. there was a path following component built into the program). To search for this component, the developer would need to know this component is a "hidden" control option of the graphical user interface (GUI). To track down this control option, the developer would need to know the key phrase "Follow Path" from GUI, and search for that. They would then find that this loads a lower control level, which uses multiple C++ classes. All this can get very time-consuming, and difficult to understand and manage.

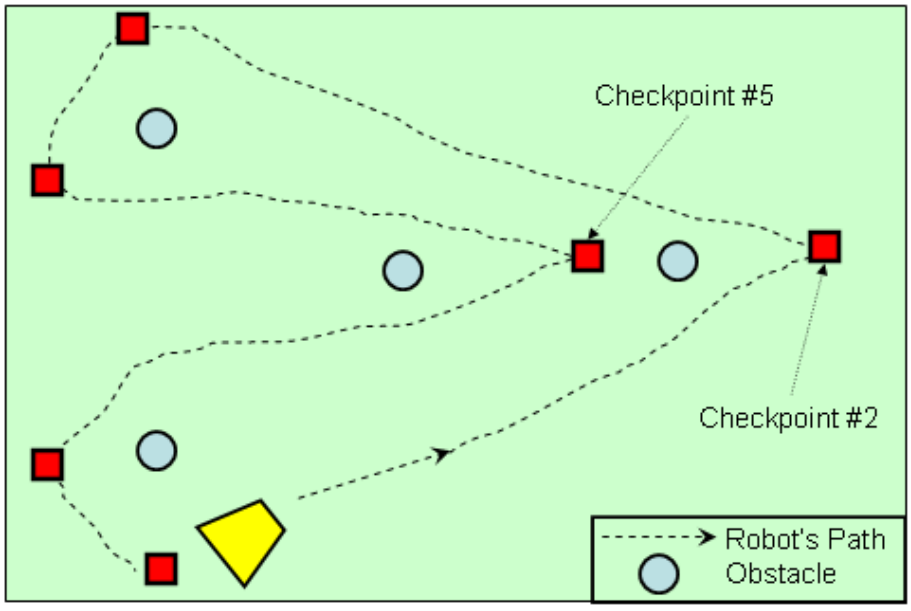


Figure 4.9: Racetrack 1 - Problem 1.

The initial design of the racetrack resulted in the robot driving a path similar to the one shown in Figure 4.9. The intent was for the robot to drive using bi-directional capabilities. For example, at checkpoint No. 2 and checkpoint No. 5, the robot was supposed to drive in reverse until it reached the following checkpoint. However, during some initial tests, the robot would often drive into the obstacle located near that checkpoint due to poor infrared communication near checkpoint No. 2. The obstacle avoidance feature was tuned so that the robot would move close to the obstacles so that the robot will not drive off the field (and out of the camera sensor's view) and so that the robot can achieve a faster time for the trial.

Racetrack change No.1

As a simple solution (now referred to as Racetrack change No.1), checkpoint No. 2 was split into two separate checkpoints (see Figure 4.10 - checkpoint No. 2 and the new checkpoint No. 3).

For this change to occur in the original C++ program, another pair of *x, y* co-ordinate values needed to be inserted into the array of checkpoints. However, in the Archangel prototype program, another state needed to be created and inserted in the XML behaviour definition (similar to Figure 4.8).

As one might expect, making this simple change to both programs was very fast (less than one minute). However, it took about three minutes to calculate the points and five more minutes to perform some additional tests. From the previous example, adding this extra state in the original C++ program was simple as adding one additional line. This assumes that the user was familiar with the original C++ program, that they knew where the array definition was, and that there would be no undesired

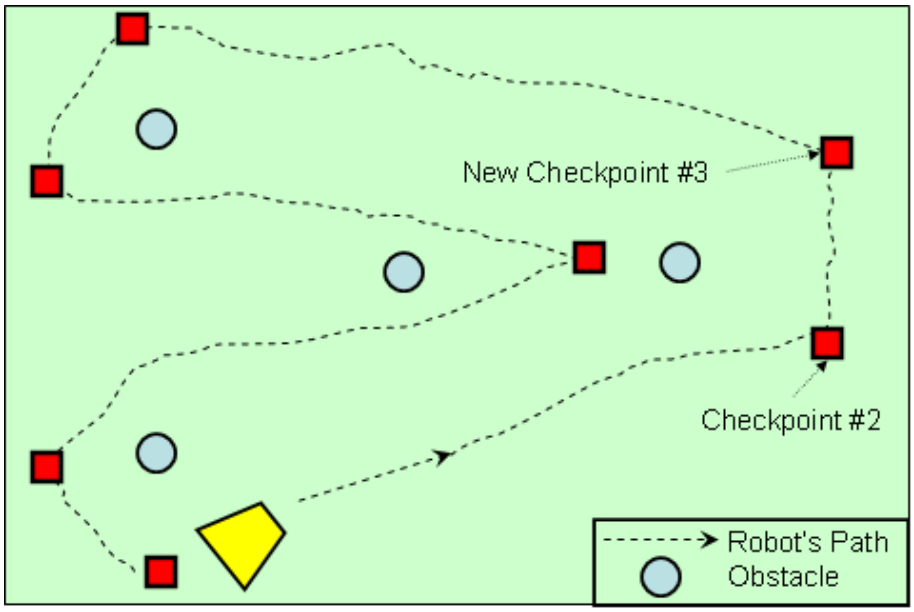


Figure 4.10: Racetrack 1 - Problem 1 Fix.

side-effects in adding a new target point. If the developer was unfamiliar with the original program, discovering this information would be a difficult task itself (as per the previous explanation that this information is implicitly distributed among several C++ classes and files).

There was slightly more work in Archangel (twelve extra lines of code were required instead of one line). Even if the developer copied the state from a template, it would require the following modifications: changing the state name, changing the name of the next state to transition to, and changing the coordinates.

Given this as evidence, one might ask why bother using the new architecture at all? The answer to this relates back to the the fundamental principles domain relevance versus domain independence of Section 2.1. The developer of the original C++ program made assumptions about the domain and task, which made code develop-

ment extremely efficient. However, a tremendous amount of flexibility was sacrificed for this efficiency. Nonetheless, for a simple change such as this, the work of adding twelve extra lines required for Archangel program, translates to mere seconds of time, which for most purposes is negligible.

Racetrack change No.2

To demonstrate the flexibility of the new architecture, the racetrack design was then changed to a different shape (see Figure 4.11).

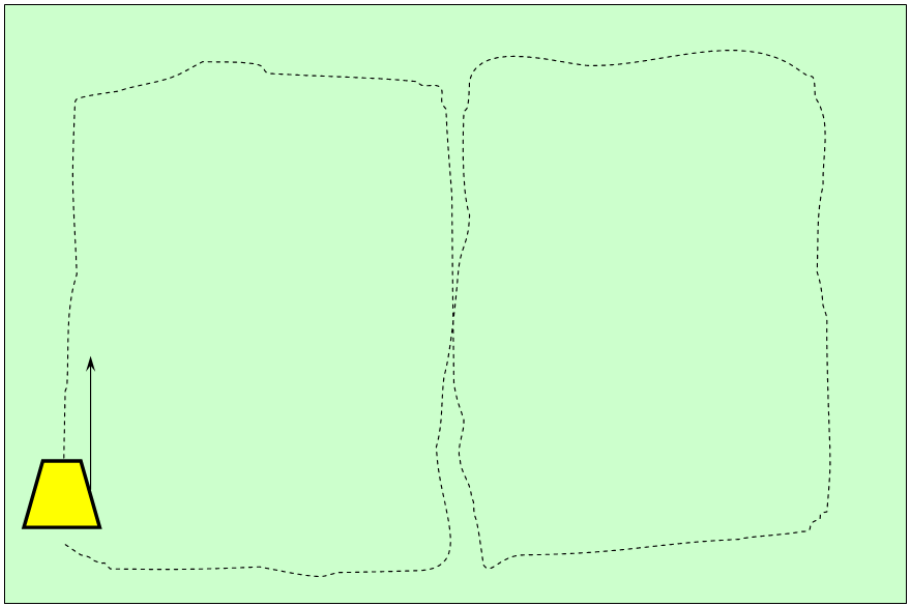


Figure 4.11: Racetrack - 2.

Making this modification (now known was Racetrack change No.2) took roughly 10 minutes to calculate the new points. Both programs required changing the co-ordinates, and the state structure was kept the same. Development for the original C++ program was slightly faster (18 minutes instead of 15 minutes). This was because there was some additional modifications (21 lines instead of 7 lines) in the XML

behaviour definition due to extra debugging aspects (e.g. drawing new target points and new paths). If these aspects were removed, or ignored, then the difference in development time would be less significant.

Nonetheless, developing for the original C++ programs were neither intuitive nor flexible. In terms of intuitiveness, the racetrack's XML definition is more understandable than a list of *x, y* coordinates since it associates a *purpose* with those coordinates. More importantly, the code written for this component was only reusable for tasks that require the robot to move in a fixed series of waypoints. If the robot was to perform some action between two specific waypoints, it is less flexible to so do in the original C++ programs. However, the behaviour engine code was designed to be reusable (i.e. the racetrack's XML behaviour definition can be modified to satisfy different constraints or swapped for other behaviours to solve a different problem).

4.3.2 Treasure Hunt Challenge

The treasure hunt challenge is different from the previous racetrack challenge in that there is no fixed path for the robot to follow. The robot must discover the path on its own. As described in Section 4.1, the treasure hunt challenge has the robot searching for "treasures" in a given space. For this task, three XML behaviour definitions were created in Archangel: *TreasureHunt*, *TreasureHuntChase* and *TreasureHuntTurn* (see Figure 4.12).

The *TreasureHunt* is the behaviour that encompasses the general behaviour for this task (see Figure 4.13). One noteworthy function it performs is that it initializes a <`target_list`>, which are the "treasures" for this task. These targets are taken

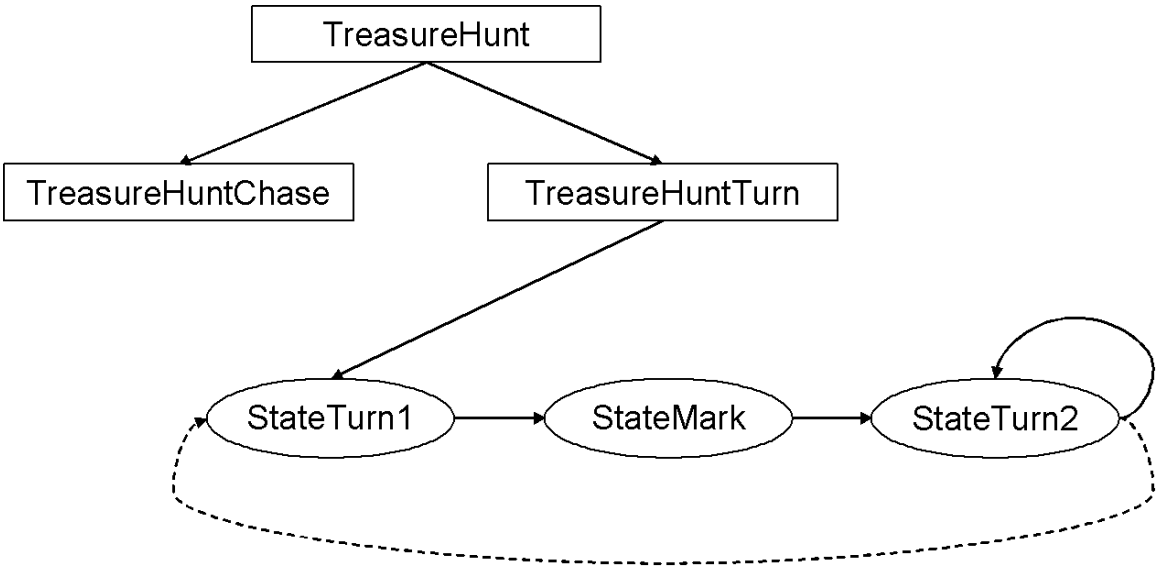


Figure 4.12: Final version of Treasure Hunt XML Tree and FSA diagram.

from the collection of tracked objects labelled as "obstacles" in the world model. Technically, these objects are not obstacles. However, the sensor system recognizes them as such. As described in Section 4.2, the sensor system is a separate component, which is effectively a "black box" (i.e. not modifiable) from the standpoint of this architecture. Thus, it is not truly important as to what tracked objects are classified as for this task. However, the potential field parameters were modified to suit this task. That is, the repelling force of the obstacles were turned off so that the path planning module will not treat them as objects to avoid.

The *TreasureHunt* also explicitly lists sub-behaviours, and it also explicitly states that the behaviour list will be used. The reward and applicability is irrelevant at this juncture, because the treasureHunt will be the only behaviour that is loaded.

TreasureHuntChase is a simple behaviour that is intended to move toward the treasures (see Figure 4.14). This behaviour is applicable when the robot is further

```
- <behaviour name="treasureHunt">
  - <init>
      <target_list ofType="obstacle" src="World::videoObjects"/>
    </init>
  - <behaviour_list>
      <behaviour_ref name="treasureHuntChase"/>
      <behaviour_ref name="treasureHuntTurn"/>
    </behaviour_list>
    <reward value="1"/>
    <applicability value="1"/>
    <execute useBehaviourList="true"/>
  </behaviour>
```

Figure 4.13: Treasure Hunt XML.

than a certain distance (e.g. fifty millimeters) away from the "closestTarget". The "closestTarget" is a position that is calculated from the World model. If it is further than a minimum distance (e.g. fifty millimeters) away from the target position, then this behaviour instructs the robot to go to the target position. The system allows for a set tolerance (e.g. forty millimeters) when determining whether a robot has reached its target point or not.

TreasureHuntTurn is a more complex behaviour that contains a FSA (see Figure 4.12 and Figure 4.15). It also has a minimum execution time of five seconds. This guarantees that the robot will stay in place (with respect to the geographical coordinates of x and y) for five seconds to fulfill the requirements of this task as specified in Section 4.1.

The applicability conditions of this behaviour are the inverse of the *TreasureHuntChase* behaviour. That is, *TreasureHuntTurn* is applicable when it is within a certain distance (e.g. fifty millimeters) of the "closestTarget". This simplifies the conflict management issue relating to this task.

```
- <behaviour name="treasureHuntChase">
    <init/>
    <reward value="1"/>
   - <applicability>
     - <condition>
       - <robot fartherThan="50">
           <reference_position reference="closestTarget"/>
           <add value="0.8"/>
         </robot>
       </condition>
    </applicability>
   - <execute>
     - <goto within="40">
         <reference_position reference="closestTarget"/>
       </goto>
    </execute>
  </behaviour>
```

Figure 4.14: Treasure Hunt Chase XML.

Initially, this behaviour was a simple behaviour that just marked the target position as completed. It was later converted to a FSA to make use of the holonomic turning capabilities (e.g. turn on the spot) of the robots.

The `StateTurn1` state is the initial state, which was added later to utilize the holonomic turning capabilities. The action in this state is to turn towards the closest target position. It will transition to the next state, `StateMark`, when the robot is truly within a minimum distance (fifty millimeters).

The `StateMark` state marks the target position completed (i.e. visited). This state then unconditionally transitions to the state `StateTurn2`.

The XML behaviour definition required less work in terms of code size than the original C++ program (e.g. 40 lines instead of 50 lines). This was because much

```
- <behaviour name="treasureHuntTurn" minExecMicroSecs="5000000">
    <init/>
    <reward value="1"/>
  - <applicability>
    - <condition>
      - <robot within="50">
          <reference_position reference="closestTarget"/>
          <add value="0.8"/>
        </robot>
      </condition>
    </applicability>
  - <execute initialState="stateTurn1" autoResetFSM="true">
    - <state name="stateTurn1">
      - <turn direction="towards">
          <reference_position reference="closestTarget"/>
        </turn>
      - <next_state name="stateMark">
        - <condition>
          - <robot within="50">
              <reference_position reference="closestTarget"/>
            </robot>
          </condition>
        </next_state>
      </state>
    - <state name="stateMark">
        <mark_complete target="closestTarget"/>
      - <next_state name="stateTurn2">
          <condition met="1"/>
        </next_state>
      </state>
    - <state name="stateTurn2">
      - <turn direction="towards">
          <reference_position reference="closestTarget"/>
        </turn>
      - <next_state name="stateTurn2">
          <condition met="1"/>
        </next_state>
      </state>
    </execute>
  </behaviour>
```

Figure 4.15: Treasure Hunt Turn XML.

of the low level implementation was abstracted using the high-level specification. In terms of development time, this task required three days to develop and test using the Archangel prototype. Much of this time was used to debug low level implementation details (not included in the code size value specified previously) and for testing. Using the original C++ program, it also took three days to develop. Thus, there was not any savings in terms of development time.

Treasure Hunt change No.1

Originally, `StateTurn1` and `StateTurn2` did not exist. However, after some debugging, state `StateTurn1` was added, which acts as a delay for the state `StateMark`. This change will henceforth be referred to as Treasure Hunt change No.1. In some initial testing scenarios, this extra state was not part of the state machine, and only the *TreasureHuntTurn* (originally named TreasureHuntMark) behaviour was loaded to test this behaviour separately. That is, the *TreasureHuntChase* behaviour was not loaded. What occurred was that the robot would mark each target (at each iteration of the control loop) as being completed (i.e. visited) even though it did not visit those target positions. This was not the desired observable actions, as the idea was to visit a target position in order to mark it as completed.

To complete the fix, `StateTurn2` was added as well. The `StateTurn2` state functions exactly the same as the `StateTurn1` state in that it turns towards the next "closestTarget". However, it was made into a leaf node, or an end state (as part of Treasure Hunt change No.1). That is, the next state to transition to is itself. The reason for this was because there were no other "safe" states to transition to – to transition to `StateTurn1` and `StateMark` would both prematurely mark target positions as com-

pleted. Since there are time constraints on the *TreasureHuntTurn* behaviour (which prevent the task manager from selecting another behaviour), performing `StateTurn2` for the remaining minimum execution time should not be detrimental to the system. These two extra states (`StateTurn1` and `StateTurn2`) enforced the idea that the system should not mark a target position as completed until the robot actually visits it. The human developer can then move the robot to each treasure manually to further test the actions of the *TreasureHuntTurn* behaviour.

A special feature of this state machine is that it will reset the state machine back to the initial state (`StateTurn1`) if the task manager switches away to another behaviour and then switches back to this behaviour. This is done by the behaviour engine and task manager accessing the history mentioned in Section 3.2.3.

This change simply required ten lines to be added in the original C++ program and required twelve lines to be added with two more lines modified in the Archangel program. Thus, more changes were needed of the Archangel program. However, timewise, Archangel required only eight minutes, whereas the original program required twenty minutes. This was because the changes in the Archangel were explicit and spatially localized. The changes required in the original program were more widespread. The developer needs to be aware of the impact of the changes and also keep track of the changes.

Treasure Hunt change No.2

The entire *TreasureHunt* (and *TreasureHuntTurn* and *TreasureHuntChase*) behaviour could have been designed as a FSA. However, splitting this up in the manner that has been done here serves several different purposes. One is that it makes debug-

ging the actions of the behaviour simpler, because the developer can debug certain actions separately to ensure they work properly. This adds some flexibility in the behaviour design. Secondly, it makes actions more explicit and distinct, which also makes things more intuitive since the developer can see exactly how a behaviour is supposed to work. Another reason is for reusability. To make behaviour definitions more flexible, it would help if it made some behaviours more reusable. From experimentation, these behaviours were determined to work correctly on Lego Mindstorm robots. However, a small modification must be made for the behaviours to perform correctly with the miniature tank robots. Due to the fact that the miniature tanks use a fixed proprietary IR protocol, there are certain actions that they can and cannot do. For example, an IR tank cannot turn on the spot. If the robot were to attempt to turn on the spot, it may move out of position. If the requirements for the task were very strict, such that the robot may not move at all from the position to pause/turn, then moving out of position should not be a possible action for the robot. Thus, any turning action in the *TreasureHuntTurn* needs to be changed to specify that the tank should not move (this modification is Treasure Hunt change No.2). Also, the *TreasureHuntChase* behaviour does not need to be modified and can be reused (the same as be said for *TreasureHunt*).

Only one line needed to be removed from the behaviour in the original C++ program to deal with this change. Six lines needed to be removed from the XML behaviour definition of Archangel. However, since these six lines are explicit and spatially localized, removing these six lines were as simple and almost as fast as removing that one line in the original C++ program. Overall time-wise, this change

was even faster in the Archangel program. This was partially because the original C++ program needed to be recompiled whereas the Archangel program did not need to be recompiled. The rest of the time difference was due to testing time.

4.3.3 Obstacle Run Challenge

The obstacle run differs from the treasure hunt in that the target positions (e.g. the treasures of the previous task) are not in fixed, discrete locations. Instead there is a zone which can have multiple target positions. Also, the robot will run back and forth between two target zones, all the while avoiding dynamic obstacles. For this task, one XML behaviour definition was needed. This behaviour has two states, one state for each zone (see Figure 4.16).

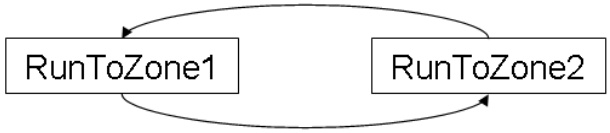


Figure 4.16: Obstacle Run state diagram.

Since the robot must avoid obstacles in this challenge, the obstacle avoidance is turned on in the XML behaviour definition. The low level controller will use some algorithm (e.g. potential fields) to perform this behaviour.

In the XML behaviour definition for the state (see Figure 4.17), the robot is instructed to move to a specific position given in absolute coordinates. This is because the path planner requires a specific target position. The primary condition to switch to the next state is that the robot must be within a certain distance (e.g. fifty millimeters) of the designated target position.

```
- <behaviour name="obstacleRun">
    <init/>
    <reward value="1"/>
    <applicability value="1"/>
  - <execute initialState="RunToZone1">
    - <state name="RunToZone1">
      - <goto>
          <control_command avoidBall="true"/>
          <absolute_position x="2640" y="760"/>
        </goto>
      - <next_state name="RunToZone2">
        - <condition>
          - <robot within="50">
              <absolute_position x="2640" y="760"/>
            </robot>
          </condition>
        </next_state>
       <!-- trigger if reach end zone 1 -->
      - <trigger_set>
        - <trigger_next_state targetName="RunToZone2">
          - <condition>
            - <robot>
                <within_rect x="2520" y="260" width="220" height="1000"/>
              </robot>
            </condition>
          </trigger_next_state>
        </trigger_set>
      </state>
```

Figure 4.17: A sample of the Obstacle Run XML.

The XML behaviour definition was originally thirty lines long, whereas the behaviour in the original C++ program is forty lines long. The behaviours in the original program are longer since their implementation requires certain functions to be implemented regardless of their use (e.g. a paint() function). The XML behaviour is less stringent in this case. Development of this task in the original C++ program took two days, whereas in the Archangel program it took merely one hour for the initial XML behaviour definition. The reason for this was during the development of the original C++ program, there were many bugs and the program was not mature enough (i.e. still in its beta testing stages). Re-development of this behaviour is

expected to be faster in the original program since it is now more mature. On the other hand, the reason why development in Archangel was so fast was this behaviour was developed along side the racetrack challenge. That is, the foundation for these two challenges (e.g. the behaviour engine) was mature enough to support the initial design of this behaviour. Given this scenario, it is difficult to compare the two since the maturity of the program at the time of developing of the behaviours needs to be considered. However, it is difficult to control the development path of any project due to the various factors that affects it (e.g. the experience levels of developers in designing and implement specific aspects).

Obstacle Run change No.1

Instructing the robot to run to a static discrete location on the field is a simple solution. If this was all there was to this state (see Figure 4.17), then the flaw should be obvious: the target position could be surrounded by obstacles (see Figure 4.18). Thus, it would be impossible for the robot to ever reach that point precisely.

However, the requirements were only that the robot reach any position within the goal area. Thus, it is not necessary for the robot to reach that point *precisely*. Changing the coordinates of this position is trivial in this architecture. However, in a dynamic environment where the obstacles can move, the continuous changing of this position by the developer is infeasible. Automatic changing of this position is not difficult: however this is not intuitive and it removes control from the behaviour level. This will make the system harder to debug.

Therefore, a better solution is to have several triggers that will be activated when the robot reaches any position within that end-zone, and have these triggers explicit

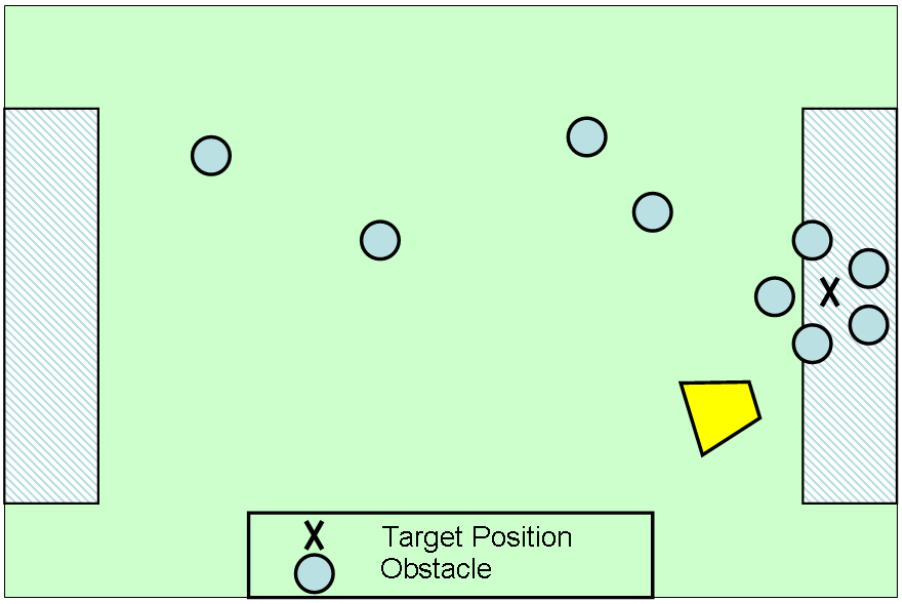


Figure 4.18: Obstacle Run - Target position surrounded scenario.

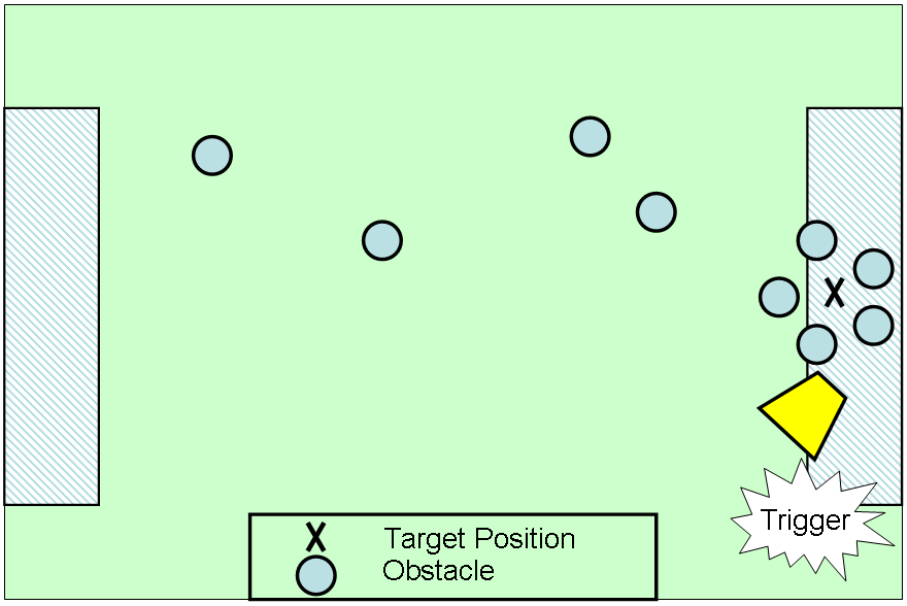
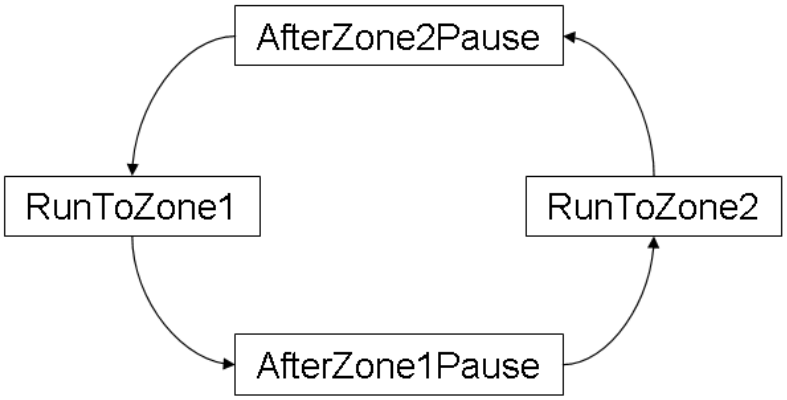


Figure 4.19: Obstacle Run - Target position surrounded scenario with trigger.

in the XML behaviour definition. This change will be referred to as Obstacle Run change No.1.

The triggers in the XML behaviour definition need to explicitly specify the next state that the behaviour system should transition to if a condition was met. The condition is when the robot is within a rectangular shaped area. The robot will attempt to wander to that surrounded target position, but may move within the goal area in its attempts (e.g. by noise in sensors or imprecision in the actuators influencing the continuous potential fields model), thus triggering the secondary condition (see Figure 4.19).

The changes in the original C++ required four additional lines. However, changes in the XML behaviour definition required eighteen lines. Development time took forty minutes using the original C++ program, whereas it two hours for the equivalent changes in the Archangel program. The reason for the difference in time was that simple trigger support is simple to implement in the original C++ program (e.g. adding a couple of conditions at key locations). At this point, trigger support needed further refinement in Archangel to support this feature. However, the trigger support is now at a point, where similar changes will be just as fast in Archangel. In terms of intuitiveness, the trigger implementation in Archangel is more intuitive since this information is made explicit. Equivalent information is still implicit in the behaviours of the original C++ program. Making these changes will require the developer to find the key locations, which is not very intuitive.




```
- <state name="AfterZone1Pause" minExecMicroSecs="5000000">
  - <goto>
      <reference_position reference="World::self"/>
    </goto>
  - <next_state name="RunToZone2">
      <condition met="true"/>
    </next_state>
</state>
```

Figure 4.20: Obstacle Run modified state diagram with sample XML addition.

Obstacle Run change No.2

To demonstrate the flexibility aspect, a new requirement was added to this task. The robot now had to stop for five seconds once it reached an end-zone before continuing. Making this change in the program with the new architecture was simple as adding two extra states within the XML behaviour definition file (See Figure 4.20). This change will be referred to as Obstacle Run change No.2.

This change took merely seconds. However a similar change in the old programs required additional variables to be defined, which required the addition of several new states. Then a more complicated decision-tree was needed to help distinguish between the states. Development for this change in Archangel took thirteen minutes

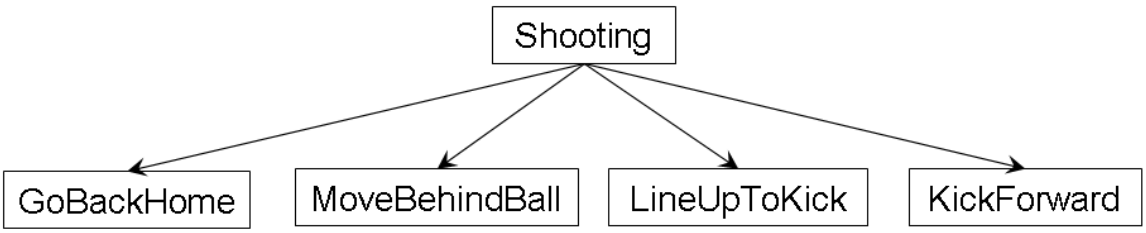
overall with eighteen additional lines, whereas the original C++ program took thirty minutes with thirty additional lines. One may argue that a simple sleep command in the C++ code should be sufficient: however, this will not send any IR commands to the robot. For the Lego robots, if no IR was received within a certain timeout (three seconds), the robot will move randomly in order to attempt to locate a stronger IR signal, making a sleep command, used alone, ineffective. In addition to the sleep statement, the command sender would need its own thread, and must transmit the stop command if the task planner does not issue a command. Even if this work was done, consider the additional requirement of reacting and avoiding obstacles. Clearly, given this constraint, a sleep command will not be sufficient. In terms of its flexibility, Archangel is more flexible and intuitive since it facilitates such scenarios.

4.3.4 Goal Scoring Challenge

The goal scoring task differs from the previous challenges in that the robot must be able to interact with another object (i.e. the ball) in the environment. For the goal scoring task, initially in the new architecture program, five separate XML behaviours were created: *Shooting*, *MoveBehindBall*, *LineUpToKick*, *KickForward*, and *GoBackHome* (see Figure 4.21).

The *Shooting* behaviour encompasses the overall behaviour for this goal scoring task. That is, it is the root node of this behaviour tree. It places the other behaviours in a behaviour list, which its <`execute`> statement explicitly states is to be used.

The initial XML behaviour definition required eighty-four lines, and took five days to implement and test. However, development of this task in the original C++




```
- <behaviour name="shooting">
  - <behaviour_list>
      <behaviour_ref name="GoBackHome"/>
      <behaviour_ref name="MoveBehindBall"/>
      <behaviour_ref name="LineUpToKick"/>
      <behaviour_ref name="KickForward"/>
    </behaviour_list>
    <reward value="1"/>
    <applicability/>
    <execute useBehaviourList="true"/>
  </behaviour>
```

Figure 4.21: Initial Goal Scoring Behaviours.

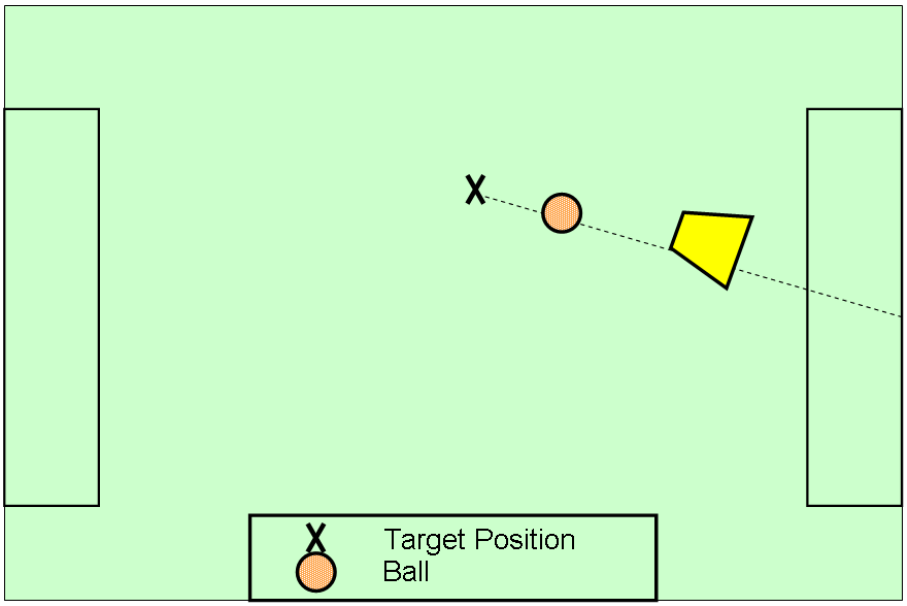


Figure 4.22: Avoiding the Ball in Goal Scoring.

program required 172 lines and took one week to implement and test. This was the fourth challenge to be developed and was one of the most difficult to develop (i.e. it took the most time out of all the challenges thus far). The reason was that it is difficult to interact with the ball (e.g. the ball might hit a bump and move in an unexpected direction). The programs were often not mature enough to handle all the unexpected scenarios that could come up. However, the XML behaviour definition provided a level of abstraction which served to hide the less informative computational information, yet made the relevant functional behaviour information explicit. This made working with the Archangel program more intuitive.

Goal Scoring change No.1

The *MoveBehindBall* behaviour had a minimum execution time of 0.2 seconds (to prevent behaviour oscillation). It was designed to bring the robot to a position behind the ball, aimed towards the center of the opponent's goal. One important scenario to avoid was having the robot between the ball and center of the opponent's goal (see Figure 4.22). In order to reach such a position, the robot may attempt to drive right through the ball, which would result in the robot moving the ball further away from the goal (or even worse, into its own goal if this was a soccer match). Thus, we should explicitly state for the robot to avoid the ball. Initially, this behaviour only had one condition, that being the ball is not in the opponent's goal. However, this condition alone caused a few problems. For example, if the ball was not found or it was not found in the field, then the execution of this behaviour would fail. That is, the robot cannot move to a position behind the ball if the position of the ball is not known. Thus, two extra conditions were required. This change will now be known as Goal

Scoring change No.1. The first additional condition was that the ball had to be found on the field. The second additional condition was that the robot had to be further than a specific distance (e.g. four hundred millimeters) away from the target position behind the ball. Adding these extra conditions required only eight more lines in the XML behaviour definition. More importantly, making these conditions explicit help reinforce the idea and motivation behind this behaviour.

The *LineUpToKick* behaviour also had a minimum execution time of 0.2 seconds. It was designed to turn the robot towards the ball for preparation of the *KickForward* behaviour. The applicability condition for this behaviour is that the robot must be within a specific distance (e.g. three hundred millimeters) of the target position (a position behind the ball facing the center of the opponent's goal) and the probability of the shot on goal is high (i.e. the robot is aligned with the ball and any point within the goal).

The *KickForward* behaviour executes for a minimum of 0.15 seconds. It was designed to kick forward. For both the Lego robots and the miniature tank robots, this meant that the robot will drive forward at top speeds for a fraction of a second. The applicability conditions for this behaviour depends on the shot on goal being very high.

Development for this change required eight lines in both the original C++ program and the Archangel program. In terms of development time, the Archangel program took nine minutes, and the original C++ program also required nine minutes. This was mostly attributed to the fact that the original program needed to be re-compiled. This modification proved to be simple enough in either programs.

Goal Scoring change No.2

The *GoBackHome* behaviour was designed to move the robot to a position behind the center line in order to be ready for the next iteration of goal scoring. To help with the debugging of this behaviour, the "home" position is explicitly stated to be drawn on the field. Initially there was only one condition for this behaviour to execute: the ball must be in the goal. However, after some further experiments, some unexpected scenarios were discovered, specifically when either the ball was found but it was not on the field, or the ball was not found anywhere in the environment. Given these two unexpected scenarios, two extra conditions were added to handle these scenarios, so that the robot will perform the *GoBackHome* behaviour in these situations as well. This change will be referred to as Goal Scoring change No.2. Making these changes was simple in this architecture. The first unexpected scenario took six additional lines in the XML behaviour definition, and the second unexpected scenario took five additional lines. Making this change in a purely C++ program required roughly the same number of line changes. However, the code was not as clean and intuitive as the XML behaviour. The results for the development time for this change was the same as the previous change, for similar reasons.

Goal Scoring change No.3

Also, during the debugging phase, it was difficult to understand why the *Kick-Forward* behaviour was not always activating when it was supposed to. Thus, the *LineUpToKick* behaviour was converted into a second state of the *MoveBehindBall* behaviour (i.e. a new XML behaviour was created from the two old XML behaviours). This change will be referred to as Goal Scoring change No.3. Figure 4.23 shows how

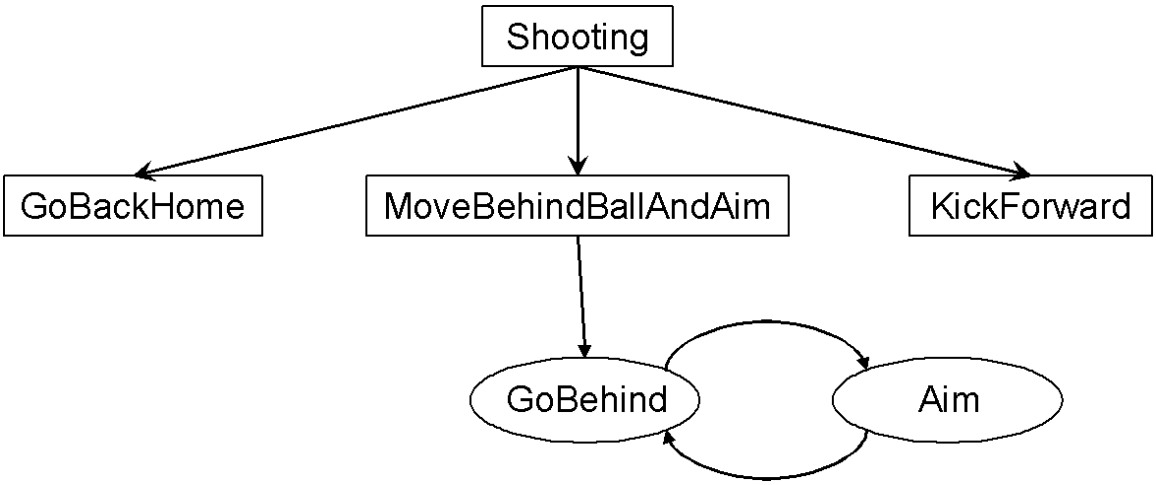


Figure 4.23: Modified Goal Scoring Behaviour.

the overall goal scoring behaviour tree has changed as a result of this: a new behaviour *MoveBehindBallAndAim* employs a simple FSA.

The `GoBehind` state encompasses all the functionality of the old *MoveBehindBall* behaviour. The condition under which to transition to the next state is that the robot must be within a specific amount (fifty millimeters) from the target position (the position behind the ball).

The `Aim` state turns the robot towards the ball, and it transitions to the next state (`GoBehind`) if it is further than a specific distance (ninety millimeters) from the target position.

The main changes in both programs involved removing behaviours and adding another behaviour. Making these changes in the original C++ program required modifications of the task planner itself, since the task planner knows only the behaviours provided when it was compiled. This is not as flexible as the Archangel architecture.

Overall, this was a larger change compared to the previous changes. It took a forty minutes to change nineteen lines in the XML behaviour definitions and fifty minutes to change twenty lines in the original C++ program. For this change, the Archangel was more intuitive, and thus the changes were performed more quickly, since it was only important, explicit information that was merged. The original C++ program required the developer to read the code to decide what code was relevant and important to be re-used.

Goal Scoring change No.4

After the modifications of Goal Scoring change No.3 were completed, another problem was discovered, in that the kick behaviour was activating too often – even when the robot was far from the ball. The solution to this problem was an additional condition in its applicability: the robot must be within two hundred millimeters of the ball. This change will be referred to as Goal Scoring change No.4.

This change required two additional lines in the original C++ program, but four additional lines in the Archangel program. However, it took only six minutes in Archangel, and nine minutes in the original C++ program. The difference in time can be attributed to the fact that changes in Archangel were at a higher specification level than that of the C++ program (and required no re-compilation).

Goal Scoring change No.5

To further demonstrate the flexibility of the Archangel architecture, another requirement was added (this is Goal Scoring change No.5). This requirement was to have the robot *back up* or drive in reverse once it kicked the ball into the goal. The purpose of this requirement was that the net had a mesh net, which made finding the

robot difficult with the global vision camera. Having the robot back up will hopefully get the robot to move away from the net and back onto the field, where it can be more easily recognized by the global vision system. To satisfy this requirement, an additional state called `backup` was added to the *kickForward* behaviours (see Figure 4.24). The *kickForward* behaviour's original kicking motion was placed in another state called `kick`. To transition from the `kick` state to the `backup` state, the ball had to be found within the goal, or the ball either had to be not found by the global vision camera. The latter proved useful in cases where the camera calibration was not precise, and the robot ended up occluding the ball from the camera in its attempts to kick the ball. The `backup` state helped move the robot back to a position where it would not occlude the ball (and give the robot a second chance to kick the ball). Each state has a minimum execution time of 70 milliseconds.

This change took fourteen additional lines in the original C++ program, but took twenty lines in Archangel. However, in terms of development time Archangel took thirteen minutes as opposed to thirty minutes needed of the original C++ program. The reason was that the changes required in the original C++ program were implicit and more widely dispersed. Making this change in Archangel was more intuitive since the relevant changes was made explicit and located in a small section within the XML behaviour definition.

4.3.5 Passing Challenge

The final challenge is the passing challenge. This challenge is different from the rest in that it requires two robots to interact with each other and with part of the

```
- <behaviour name="kickForward" minExecMicroSecs="150000">
    <init/>
    <reward value="1"/>
  - <applicability>
    - <condition>
      - <robot within="180">
          <reference_position reference="World::ball"/>
          <add value="0.2"/>
        </robot>
      - <shot_on_goal checkFacingGoal="true" probability="high">
          <add value="0.25"/>
        </shot_on_goal>
      </condition>
    </applicability>
  - <execute initialState="kick" autoResetFSM="true">
    - <state name="kick" minExecMicroSecs="70000">
        <kick type="forward"/>
      - <next_state name="backup">
        - <condition>
            <ball isFound="false"/>
          </condition>
        </next_state>
      - <trigger_set>
        - <trigger_next_state targetName="backup">
          - <condition>
            - <ball>
                <within_rect x="2710" y="435" width="30" height="650"/>
              </ball>
            </condition>
          </trigger_next_state>
        </trigger_set>
      </state>
    - <state name="backup" minExecMicroSecs="70000">
        <kick type="reverse"/>
      - <next_state name="kick">
          <condition met="1"/>
        </next_state>
      </state>
    </execute>
  </behaviour>
```

Figure 4.24: New KickForward XML behaviour definition with *backup* state.

environment (i.e. the ball). For the passing challenge, six XML behaviours were created in the Archangel program. Each robot had three behaviours (see Figure 4.25). The main differences between the behaviours of robot1 and robot2 were due to the coordinates on the field to which each of the robot had to move. Using similar but separate behaviours also help in customizing the control program for each robot. For example, if one robot is more sensitive to motor commands (i.e. sometime twitches greatly from the odd command), the control behaviour for that robot can specify to have the robot move further away from the ball when the robot needs to move around the ball (so as not to accidentally bump it).

The first behaviour was the overall tree behaviour for the robot, which was called *Robot1Passing* for robot1 (and *Robot2Passing* for robot2). It had two child behaviours called *Robot1PassingGoBackHome*, and *Robot1PassingDrill*. The second robot had similarly named behaviours.

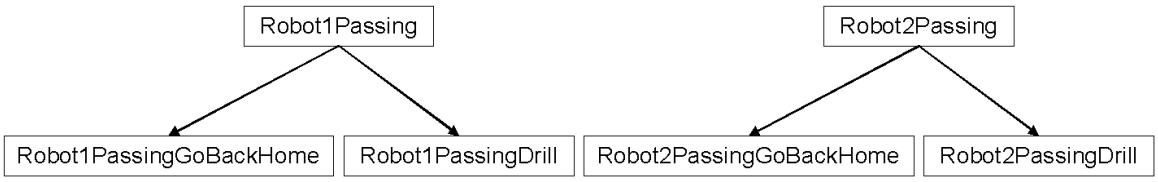


Figure 4.25: Robot1 and Robot2 Behaviour Trees.

Robot1PassingGoBackHome instructs the robot to go to the "home" location (a specific position on the field specified by absolute coordinates) on its side of the field, which is in one corner position. This behaviour is applicable when the ball is not found or not on the robot's side of the field.

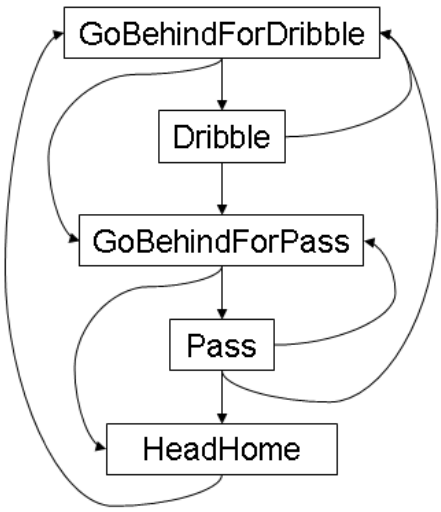


Figure 4.26: Passing Drill.

The *Robot1PassingDrill* behaviour has several internal states. These internal states can be remodelled as a behaviour tree instead as well, so from this point onwards the concept of state and behaviour can be used interchangeably. These internal states are as follows (see Figure 4.26):

- *GoBehindBallForDribble*;
- *Dribble*;
- *GoBehindBallForPass*;
- *Pass* and;
- *HeadHome*.

The initial state is the *GoBehindBallForDribble* behaviour, which instructs robot to go to a target position behind the ball to prepare for the dribbling procedure using an absolute focus point on the other quarter of the field while avoiding the ball (i.e. the

robot needs to drive around the ball) (see Figure 4.27 for the XML description of this state). In order to transition to the next state, *Dribble*, the robot must be within a specific distance (fifty millimeters) from that target position. In addition, there is a trigger that will take the robot to the *GoBehindBallForPass* behaviour in situations where the ball somehow ended on the other quarter of the field. This scenario can happen if the robot accidently bumped the ball (whether the ball was in a static position or was still moving), or if there was third party intervention (e.g. a human operator moved the ball manually).

```
<!-- ball is this half of the field-->
- <state name="goBehindBallForDribble">
  - <goto>
      <control_command avoidBall="true"/>
      <relative_abs_focus_position offsetPos="farBehind"
      reference="World::ball" focusPoint.x="600" focusPoint.y="200"/>
    </goto>
  - <next_state name="dribble">
    - <condition>
      - <robot within="50">
          <relative_abs_focus_position offsetPos="farBehind"
          reference="World::ball" focusPoint.x="600" focusPoint.y="200"/>
        </robot>
      </condition>
    </next_state>
   <!-- trigger if ball reach next quadrant -->
  - <trigger_set>
    - <trigger_next_state targetName="goBehindBallForPass">
      - <condition>
        - <ball isFound="true">
            <within_rect width="1370" x="0" y="0" height="600"/>
          </ball>
        </condition>
      </trigger_next_state>
    </trigger_set>
</state>
```

Figure 4.27: GoBehindBallForDribble XML state.

The initial XML behaviour definition for this task required 322 lines and the original C++ program need just as many lines (320 lines). However, the difference in development time between the two programs was great. It took three days in the Archangel program, whereas this development in the original C++ program took two weeks. Part of this difference was due to the difficulty in working with the ball and with more than one robot. However, much of the difference was also due to the intuitiveness of the Archangel architecture itself, and more specifically the XML behaviour definition methodology.

Passing change No.1

The *Dribble* behaviour is designed to dribble the ball to that absolute focus point on the other quarter of the field (see Figure 4.28 for the XML description of this state). Alternatively, the behaviour can be changed to move towards the ball as well (passing change No.1). However, this method explicitly represents the destination to which the robot intends to move the ball. This behaviour also has a trigger to move back to the *GoBehindBallForDribble* behaviour if the robot loses possession of the ball, which happens often to robots without dribble bars (mechanisms which spin in such a fashion as to create an attracting force to the ball).

Performing the changes for passing change No.1 required only one line to be modified in the Archangel program, but four lines in the original C++ program. Development time for this change took fifteen minutes in the Archangel program and twenty-three minutes in the original C++ program. The reason for the time difference and size of the changes was because the original C++ program required more low level involvement.

```
- <state name="dribble">
  - <goto>
      <control_command avoidBall="false"/>
      <absolute_position x="600" y="200"/>
    </goto>
  - <next_state name="goBehindBallForPass">
    - <condition>
      - <ball isFound="true">
          <within_rect width="1370" x="0" y="0" height="600"/>
        </ball>
      </condition>
    </next_state>
  - <!--
     Go back behind ball if we're no longer pushing ball
    -->
  - <trigger_set>
    - <trigger_next_state targetName="goBehindBallForDribble">
      - <condition>
        - <robot>
            <within_rect width="1370" x="0" y="0" height="600"/>
          </robot>
          <!-- AND -->
        - <ball>
            <within_rect width="1370" x="0" y="600" height="920"/>
          </ball>
        </condition>
      </trigger_next_state>
    </trigger_set>
</state>
```

Figure 4.28: Dribble XML state.

The *GoBehindBallForPass* behaviour is similar to the *GoBehindBallForDribble* behaviour except that the focus point is an absolute position on the other side of the field (see Figure 4.29 for the XML description of this state). Its default next state is the *Pass* behaviour, but it also has a trigger to the *HeadHome* state to handle an unexpected scenario: that of the ball ending up on the other half of the field (e.g. by the robot accidently bumping it there, or through human intervention).

```
- <state name="goBehindBallForPass">
  - <goto>
      <relative_abs_focus_position offsetPos="behind"
      reference="World::ball" focusPoint.x="2740" focusPoint.y="200"/>
    </goto>
  - <next_state name="pass">
    - <condition>
      - <robot within="50">
          <relative_abs_focus_position offsetPos="behind"
          reference="World::ball" focusPoint.x="2740"
          focusPoint.y="200"/>
        </robot>
      </condition>
    </next_state>
   <!-- trigger if ball crosses the line -->
  - <trigger_set>
    - <trigger_next_state targetName="headHome">
      - <condition>
        - <ball isFound="true">
            <within_rect width="1370" x="1410" y="0" height="1520"/>
          </ball>
        </condition>
      </trigger_next_state>
    </trigger_set>
  </state>
```

Figure 4.29: GoBehindBallForPass XML state.

The *Pass* behaviour is similar to the *Dribble* behaviour in that it moves the ball towards an absolute position (however on the other half of the field). Its default next state is the *HeadHome* behaviour. However, there are two triggers to assist in unexpected scenarios. The first trigger helps transition back to the *GoBehindBall-ForPass* behaviour in situations where the robot loses possession of the ball and the ball remains on the current quarter of the field. The second trigger transitions to the *GoBehindBallForPass* behaviour and handles situation where the ball ends up back

```
- <state name="pass">
  - <goto>
      <absolute_position x="2740" y="200"/>
    </goto>
  - <next_state name="headHome">
    - <condition>
      - <ball isFound="true">
          <within_rect width="1370" x="1410" y="0" height="1520"/>
        </ball>
      </condition>
    </next_state>
  - <!--
     Go back behind ball if we're no longer pushing ball
    -->
  - <trigger_set>
    - <trigger_next_state targetName="goBehindBallForPass">
      - <condition>
        - <robot>
            <within_rect width="1370" x="1370" y="0" height="1520"/>
          </robot>
         <!-- AND -->
        - <ball>
            <within_rect width="1370" x="0" y="0" height="1520"/>
          </ball>
        </condition>
      </trigger_next_state>
    </trigger_set>
  </state>
```

Figure 4.30: Pass XML state.

in the previous quarter of the field and the robot must perform the dribbling actions again.

Passing change No.2

The *HeadHome* behaviour is just a redundant behaviour added in case the *Robot1PassingGoBackHome* was not loaded (passing change No.2). This is useful if the designer was testing *Robot1PassingDrill* separately. This redundancy also makes the system more robust against failures.

Making this change required twelve extra lines (per robot) in the Archangel program but only eight extra lines in the original C++ program. Both these changes took one hour in implementation and test (more so with the latter). The difference in lines of modification is artificial in this case as XML requires ending tags, which were placed on separate lines for readability.

Passing change No.3

Some difficulties that arose for this task were bad IR communication and sensor (vision) data in certain areas of the field, and unexpected movements (e.g. the robot running into the ball when it was not supposed to). In one scenario, the home position of the robot was near a corner. The corner position was chosen because it was deemed more likely that the ball would roll in front of the robot, and ensures that the ball will never get behind it. However, much of the behaviour was written prior to actual field tests. Upon the field tests, parts of the field were discovered to receive poor IR communication. In certain scenarios of no IR reception, the Lego robot may move randomly (or spin on the spot) to attempt to locate an IR signal. This can move the robot out of position, or may bring the robot to an undesirable position. Using this architecture, it was trivial to move this home position further away from the corner (passing change No.3), which improved the robot's performance (i.e. it will receive IR signals and will not move uncontrollably, spinning away seeking IR signals). Performing this modification in the old program is not as intuitive for a new developer.

This was a trivial change in both programs (i.e. only one line needed to be modified per robot). However, development time was faster using Archangel because of the

same reasons mentioned previously, relating to the higher layer of abstraction in the XML behaviours.

Passing change No.4

In another scenario, when the robot was trying to get around the ball, it often ran into it instead. Upon closer inspection of this behaviour, it was discovered that the behaviour did not call for the robot to avoid the ball. Thus, a control command was added for the robot to avoid the ball (passing change No.4). This helped improve performance, in that the robot was not running into the ball as often. Making such a feature explicit greatly improved debugging of this behavior. Otherwise, to debug this problem, the developer would need to trudge through several lines of C++ code to ensure that the potential fields were acting correctly to push the robot away from the ball. Even having done this, it would still be difficult to know when this pushing force will execute. Having this aspect explicit is much more intuitive. As further optimization of this aspect of avoiding the ball, the behind ball position was changed to be further back. This also improved the robot's performance in avoiding the ball. Using this architecture made this change easier and more intuitive: the value of the setup position could simply be changed from *behind* to *farBehind*. Using these descriptive values are much more intuitive in that they are easier to debug and to change. These descriptive values represent a range of values as opposed a single discrete value, which allows for some flexibility depending on the precision provided by the sensors and actuators, and also depending on the precision required by task or domain.

Both these changes requires only the addition of one extra line. Overall, development time was somewhat faster using the Archangel program (ie. ten vs. fifteen minutes). However, the difference could be greater if one was to factor in the process of diagnosing that this was the problem. It was easy to notice this problem in the XML behaviour definition, since this relevant information was made explicit. The lack of this control command to avoid the ball was easier to see. However, in the C++ program, this aspect is implicit in the code and one can easily overlook this change if one was not familiar with the program itself.

Passing change No.5

Another problem that was encountered using this behaviour was that the robot sometimes missed the ball when it was far behind it. Based on the assumption that the setup point was too far for the required dribble, the destination to dribble to (which was defined as previously stated an absolute position) was changed. This change (passing change No.5) fixed this particular problem.

As further general optimization for this task, in the *passingGoBackHome* behaviour, originally the orientation was not set. This meant that the robot could have been facing any direction. For the Lego robot, that was not making good use of the robot's holonomic driving capabilities. Thus, including the orientation in the behaviour allowed the robot to get into a better position (also part of passing change No.5) to anticipate the next behaviour (e.g. *goBehindBallForDribble*). This made for a much more effective passing play.

The changes of passing change No.5 required trivial amounts of work (i.e. two or four lines needed to be modified in either program). Development time for these

changes was somewhat comparable (e.g. twenty-five vs. fifteen minutes). Much of the reasoning behind these difference were same as those of passing change No.3 (i.e. the higher layer of abstraction methodology argument).

4.4 Evaluation Summary

The following tables, Table 4.1 and 4.2, summarize the results described in this chapter. The first table, Table 4.1, describes roughly the amount of "code" (in lines of code) that needed to be added or modified for each of the five challenges (and modifications of those challenges). The second table, Table 4.2, describes the time it took to implement the initial code and changes relating to the five challenges. The next chapter further explain these results and draw conclusions of this thesis research.

Results summary

	Original C++ Client	**Archangel**
Code size		
Racetrack original design	50 lines	88 lines of XML
Racetrack change No. 1	added 1 line	edited 19 lines
Racetrack change No. 2	modified 7 lines	modified 21 lines
Treasure Hunt original design	50 lines	40 lines of XML
Treasure Hunt change No. 1	edited 10 lines	edited 14 lines
Treasure Hunt change No. 2	edited 1 line	remove 6 lines
Obstacle Run original design	40 lines	30 lines of XML
Obstacle Run change No. 1	added 4 lines	Add 18 lines of XML
Obstacle Run change No. 2	edited 30 lines	edited 18 lines
Goal Scoring original design	172 lines	84 lines of XML
Goal Scoring change No. 1	add 8 lines	add 8 lines
Goal Scoring change No. 2	edited 10 lines	add 11 lines
Goal Scoring change No. 3	edited 20 lines	edited 19 lines
Goal Scoring change No. 4	modify 2 lines	add 4 lines
Goal Scoring change No. 5	modify 14 lines	add 20 lines
Passing original design	320 lines	322 lines of XML
Passing change No. 1	modify 4 lines (x2)	modify 1 line (x2)
Passing change No. 2	modify 8 lines	added 12 lines (x2)
Passing change No. 3	modify 1 line (x2)	modify 1 line (x2)
Passing change No. 4	modify 1 line (x2)	add 1 line (x2)
Passing change No. 5	modify 3 line (x2)	modify 2 line (x2)

Table 4.1: Code size.

Results summary - part 2

	Original C++ Client	**Archangel**
Development Time		
Racetrack original design	7 days	4 days
Racetrack change No. 1	8 minutes	8 minutes
Racetrack change No. 2	15 minutes	18 minutes
Treasure Hunt original design	3 days	3 days
Treasure Hunt change No. 1	20 minutes	8 minutes
Treasure Hunt change No. 2	6 minutes	4 minutes
Obstacle Run original design	2 days	1 hour
Obstacle Run change No. 1	40 minutes	2 hour
Obstacle Run change No. 2	30 minutes	10 minutes
Goal Scoring original design	7 days	5 days
Goal Scoring change No. 1	10 minutes	7 minutes
Goal Scoring change No. 2	9 minutes	7 minutes
Goal Scoring change No. 3	50 minutes	40 minutes
Goal Scoring change No. 4	9 minutes	6 minutes
Goal Scoring change No. 5	30 minutes	13 minutes
Passing original design	14 days	3 days
Passing change No. 1	23 minutes	15 minutes
Passing change No. 2	1 hour	1 hour
Passing change No. 3	10 minutes	6 minutes
Passing change No. 4	15 minutes	10 minutes
Passing change No. 5	25 minutes	15 minutes

Table 4.2: Development Time.

Chapter 5

Conclusions and Future Work

Evaluating an architecture is a difficult task, as described in the previous chapter. It was difficult to evaluate the architecture based on quantitative measures since terms such as flexibility, extensibility, and intuitiveness are very subjective. Arkin may have described it best when he pointed out evaluation between architectures is more a question of efficiency that computability [15]. That is, one may architecture may work better than another depending on the task, domain, and people involved. Grounding this research in a domain such as robotic soccer helped in this matter. Robotic soccer was chosen as the domain because of the strong international research support, and because a robotic soccer player represents common skills required of an intelligent mobile robot as described in Section 1.1. Through a series of challenges, the advantages of using the Archangel architecture is shown to have great potential.

From the results described in the previous chapter, there were little savings in code size (lines of code) in using the Archangel program compared to the original C++ program. Quite often, the code size increases in the Archangel program. However,

it is difficult to use code size as a measurement since not all parts of the XML code are functional: rather, they exist for validity and readability (e.g. ending tags are placed on separate lines for readability). Also, the organization of the code is not portrayed using a single value such as "lines of code". For example, the functional code (the code that performs the necessary functions to complete the task) may be too far distributed and the developer must spend a lot of time simply finding all the different parts of the code. In general, the Archangel architecture saved the developer a lot of time due to the higher layer of abstraction provided by the XML behaviour definitions. These XML behaviour definitions just needed to be reloaded, whereas the behaviours in the original C++ client program required a recompilation of the program.

However, development time is an imperfect measurement as well because various factors affect this time that are difficult to control. The development of the XML description tags and the behaviour were developed in tandem on the basis of necessity (as described in the previous chapter). This was done to keep the set of XML tags to a minimum, in order to stay intuitive and understandable so not to overwhelm developers. Because of this, the development of the XML behaviours took longer than might actually have been necessary. Had the behaviour engine (and supporting XML infrastructure) been completed prior to testing the behaviours, development time would be significantly shortened. For example, the obstacle run was developed right after the XML foundation was set in place for the racetrack challenge. Since the obstacle run is similar enough to the racetrack, significant time was saved since very little (lengthy and time-consuming) changes were needed of the behaviour engine.

Thus, the obstacle run was done in one hour as opposed to days. With the behaviour engine completed and the XML foundation in place, developing new behaviours should take significantly less time than that which was initially reported for each of the challenges.

However, in general, working with the Archangel program is more intuitive and flexible since relevant information is made explicit. Also, the higher layer of abstraction provided by the XML behaviour definitions allowed for increased flexibility. It allowed developers to focus more on the relevant high-level specification changes, and less on the lower-level changes, which saves time in many of the sample changes.

In conclusion, the Archangel architecture was able to meet the goals outlined in the beginning of this research. The overall goals were for it to be intuitive, flexible, adaptable, and extensible. The flexibility of this architecture was shown by the implementation on two types of simple robots as described in Section 4.2. One of the greatest strengths of this architecture is the behaviour system. Designing behaviours was made simple and intuitive using the explicit representation. This explicit representation made relevant information easy to notice and find, so that adding, removing and modifying behaviours is simple to do (i.e. it is flexible and extensible to suit different tasks). From several challenges/tasks from Section 4.1, the evaluation of the implementation of the architecture from Section 4.3 demonstrates how easy it was to add new behaviours, remove old behaviours, and modify existing behaviours to suit new requirements and unexpected scenarios.

Adding behaviours into the system was simple as writing another state or XML definition file. For example, as shown in Section 4.3.3, the obstacle run challenge.

Also, incorporating this addition into the existing system was made easy as adding one additional line in the parent behaviour. Conversely, removing behaviours was as simple as removing one line from the parent XML behaviour definition (e.g. as shown in Section 4.3.4, with the goal-scoring challenge). Modifying existing behaviours was easier than previous control client programs that require the programmer to re-compile the program.

Having the behaviours loosely coupled in its own XML behaviour definition files significantly simplifies the testing and debugging stage. It allows the programmer to test and debug each behaviour separately before testing all the behaviours as a whole. This allows developers to find errors faster and also design tests (i.e. test cases) to find specific scenarios where systems succeeds or fails.

In summary, as originally stated in Chapter 1, the main contributions of this thesis are:

- a survey of existing architectures;
- the design of a new intuitive and flexible architecture;
- a prototype implementation of the proposed architecture and;
- an evaluation study of the proposed architecture (via the implementation).

Chapter 2 described the existing robotic architectures and Chapter 3 described the proposed architecture, Archangel. With all the architectures presented in Chapter 2, it should be clear that Archangel is a unique architecture that has its own niche among robotic architectures. The implementation of the architecture required many hours of work, but the benefits of the architecture justifies all the effort.

For future work, there are a few ways to improve on this architecture and support its claims. As mentioned in the previous chapter, I played the role as the developer that gathered these results. That being the case, some bias may have affected the results. Some factors helped minimize this bias, specifically my familiarity with both programs. That is, results would have been extremely skewed (in favour of Archangel) if I was completely unfamiliar with the original C++ program.

As noted at the beginning of this thesis, formal user/case studies can be performed to further validate the intuitive, flexibility and usefulness aspect of this Archangel architecture. If the participants were unfamiliar with both of these programs, it should also reduce any bias that exists. However, this will require additional resources (in terms of time, people, and funding) which is beyond the scope of this Master's thesis.

To improve this architecture, it can be further ported to other systems and robots. This can potentially help extend the set of explicit representation in order to make it more flexible to use. This also supports the claim that the overall architecture is extensible.

Nonetheless, there are a large set of possibilities for further extensibility. Much of these possibilities will depend on the domain and tasks required of mobile robot. However, the Archangel architecture is flexible and intuitive enough to meet these challenges.

Appendix A

Archangel DTD

```
<!ELEMENT behaviour (init?, draw_env?, behaviour_list?,
                     reward?,applicability,execute)>

<!ATTLIST behaviour name ID #REQUIRED
                     minExecMicroSecs CDATA #IMPLIED>

<!ELEMENT init (target_list)?>

<!ELEMENT draw_env (pen | line | rect )* >

<!ELEMENT behaviour_list (behaviour_ref)*>

<!ELEMENT behaviour_ref EMPTY>
<!ATTLIST behaviour_ref name ID #REQUIRED>

<!ELEMENT reward EMPTY>
<!ATTLIST reward value CDATA #IMPLIED>

<!ELEMENT applicability (condition)*>
<!ATTLIST applicability value CDATA #IMPLIED>

<!ELEMENT execute (state | goto | kick | turn)* >
<!ATTLIST execute initialState IDREF #IMPLIED
                  autoResetFSM ( true | false | on | off | 1 | 0 ) "false"
                  useBehaviourList ( true | false ) "false">
```

```
<!ELEMENT target_list EMPTY>
<!ATTLIST target_list src (World::videoObjects) #REQUIRED
                      ofType (obstacle) #IMPLIED
                      updatable ( true | false ) #IMPLIED
                      sortBy ( distanceAscending | distanceDescending |
                               unsorted ) #IMPLIED>

<!ELEMENT pen EMPTY>
<!ATTLIST pen colour ( white | black | red | darkRed | green |
                       darkGreen | blue | darkBlue | cyan |
                       darkCyan | magenta | darkMagenta | yellow |
                       darkYellow | gray | darkGray |
                       lightGray) #REQUIRED>

<!ELEMENT line EMPTY>
<!ATTLIST line x1 CDATA #REQUIRED
               y1 CDATA #REQUIRED
               x2 CDATA #REQUIRED
               y2 CDATA #REQUIRED>

<!ELEMENT rect EMPTY>
<!ATTLIST rect x CDATA #REQUIRED
               y CDATA #REQUIRED
               width CDATA #REQUIRED
               height CDATA #REQUIRED>

<!ELEMENT state ( (draw_env | goto | kick | turn | mark_complete)*,
                  next_state, (trigger_set)? ) >

<!ATTLIST state name ID #REQUIRED
                minExecMicroSecs CDATA #IMPLIED>

<!ELEMENT goto ( ( absolute_position | reference_position |
                relative_ref_focus_position |
                relative_abs_focus_position ), control_command? ) >

<!ATTLIST goto within CDATA #IMPLIED>

<!ELEMENT control_command EMPTY>
<!ATTLIST control_command avoidBall
    ( true | false | on | off | 1 | 0 ) #REQUIRED>
```

```
<!ELEMENT absolute_position EMPTY>
<!ATTLIST absolute_position x CDATA #REQUIRED
                            y CDATA #REQUIRED
                            orientation CDATA #IMPLIED >

<!ELEMENT reference_position EMPTY>
<!ATTLIST reference_position reference
    ( closestTarget |
      World::self | World::ball |
      World::closestObstacle |
      World::closestRobot |
      World::centerField |
      World::ourKickersKickoffPosition |
      World::ourGoalkeepersKickoffPosition |
      World::ourGoalPost1 |
      World::ourGoalPost2 |
      World::theirGoalPost1 |
      World::theirGoalPost2 |
      World::ourGoalCenter |
      World::theirGoalCenter ) #REQUIRED>

<!ELEMENT relative_ref_focus_position EMPTY>
<!ATTLIST relative_ref_focus_position reference
    ( closestTarget |
      World::self | World::ball |
      World::closestObstacle |
      World::closestRobot |
      World::centerField |
      World::ourKickersKickoffPosition |
      World::ourGoalkeepersKickoffPosition |
      World::ourGoalPost1 |
      World::ourGoalPost2 |
      World::theirGoalPost1 |
      World::theirGoalPost2 |
      World::ourGoalCenter |
      World::theirGoalCenter ) #REQUIRED
    offsetPos CDATA #REQUIRED
    focusPoint ( closestTarget |
      World::self | World::ball |
      World::closestObstacle |
      World::closestRobot |
      World::centerField |
```

```
      World::ourKickersKickoffPosition |
      World::ourGoalkeepersKickoffPosition |
      World::ourGoalPost1 |
      World::ourGoalPost2 |
      World::theirGoalPost1 |
      World::theirGoalPost2 |
      World::ourGoalCenter |
      World::theirGoalCenter ) #REQUIRED>

<!ELEMENT relative_abs_focus_position EMPTY>
<!ATTLIST relative_abs_focus_position reference
    ( closestTarget |
      World::self | World::ball |
      World::closestObstacle |
      World::closestRobot |
      World::centerField |
      World::ourKickersKickoffPosition |
      World::ourGoalkeepersKickoffPosition |
      World::ourGoalPost1 |
      World::ourGoalPost2 |
      World::theirGoalPost1 |
      World::theirGoalPost2 |
      World::ourGoalCenter |
      World::theirGoalCenter ) #REQUIRED
    offsetPos CDATA #REQUIRED
      focusPoint.x CDATA #REQUIRED
      focusPoint.y CDATA #REQUIRED>

<!ELEMENT kick EMPTY>
<!ATTLIST kick type (forward | straight | reverse |
                     spinKickLeft | spinKickRight) #REQUIRED>

<!ELEMENT turn ( absolute_position | reference_position |
                 relative_ref_focus_position |
                 relative_abs_focus_position ) >
<!ATTLIST turn direction ( towards | awayFrom ) #REQUIRED>

<!ELEMENT mark_complete EMPTY>
<!ATTLIST mark_complete target (closestTarget) #REQUIRED>

<!ELEMENT next_state (#PCDATA | condition )* >
<!ATTLIST next_state name IDREF #REQUIRED>
```

```
<!ELEMENT condition (#PCDATA | robot | ball | shot_on_goal )* >
<!ATTLIST condition met ( true | false | 1 | 0 ) "false">

<!ELEMENT robot ( ( absolute_position | reference_position |
                    relative_ref_focus_position |
                    relative_abs_focus_position )?,
                    ( within_rect | not_within_rect )?,
                    ( add | subtract)? ) >
<!ATTLIST robot within CDATA #IMPLIED
                fartherThan CDATA #IMPLIED>

<!ELEMENT ball ( ( absolute_position | reference_position |
                    relative_ref_focus_position |
                    relative_abs_focus_position )?,
                    ( within_rect | not_within_rect )?,
                    ( add | subtract)? ) >
<!ATTLIST ball isFound ( true | false | 1 | 0 ) #IMPLIED
               within CDATA #IMPLIED
               fartherThan CDATA #IMPLIED>

<!ELEMENT within_rect EMPTY>
<!ATTLIST within_rect x CDATA #REQUIRED
                      y CDATA #REQUIRED
                      width CDATA #REQUIRED
                      height CDATA #REQUIRED>
<!ELEMENT not_within_rect EMPTY>
<!ATTLIST not_within_rect x CDATA #REQUIRED
                          y CDATA #REQUIRED
                          width CDATA #REQUIRED
                          height CDATA #REQUIRED>

<!ELEMENT add EMPTY>
<!ATTLIST add value CDATA #REQUIRED>

<!ELEMENT subtract EMPTY>
<!ATTLIST subtract value CDATA #REQUIRED>

<!ELEMENT shot_on_goal (add | subtract)?>
<!ATTLIST shot_on_goal facingGoal ( true | false | 1 | 0 ) #REQUIRED
                       probability CDATA #REQUIRED>
```

```
<!ELEMENT trigger_set ( trigger_next_state |
                        trigger_child_behaviour )+ >

<!ELEMENT trigger_next_state (condition)>

<!ATTLIST trigger_next_state targetName IDREF #REQUIRED>

<!ELEMENT trigger_child_behaviour (condition)>

<!ATTLIST trigger_child_behaviour targetName CDATA #REQUIRED>
```

Bibliography

[1] 74.406 - Intelligent Mobile Robotics. http://www4.cs.umanitoba.ca/%7Ejacky/Teaching/Courses/74.406-Intelligent-Mobile-Robotics/current/index.php.

[2] American Association for Artificial Intelligence (AAAI). http://www.aaai.org/.

[3] Doraemon: Robocup Video Server. http://sourceforge.net/projects/robocup-video/.

[4] E-League at Columbia University. http://agents.cs.columbia.edu/eleague/.

[5] National Institute of Standards and Technology (NIST). http://www.nist.gov/.

[6] Researchers of the E-League Project. http://agents.cs.columbia.edu/eleague/people.php.

[7] Sony Global - AIBO Global Link. http://www.sony.net/Products/aibo/.

[8] J.S. Albus. Outline for a theory of intelligence. In *IEEE Transactions on Systems, Man and Cybernetics*, volume 21, pages 473–509, May–June 1991.

[9] J.S. Albus, R. Lumia, and H.G. McCain. NASA/NBS standard reference modle for telerobot control system architecture (NASREM). Technical Report 1235, National Bureau of Standards, 1986.

[10] John Anderson, Jacky Baltes, Doug Cornelson, Terry Liu, Clint Stuart, and Adam Zilkie. The University of Manitoba ULeague team. In *Proceedings of the RoboCup Symposium*, Padova, Italy, July 2003.

[11] John Anderson, Jacky Baltes, and Terry Liu. Robobisons 2004. In Daniele Nardi, Martin Riedmiller, , and Claude Sammut, editors, *The Seventh RoboCup Competitions and Conferences*, Berlin, 2005. Springer Verlag.

[12] John Anderson, Jacky Baltes, Brian McKinnon, Terry Liu, Paul Furgale, and Shawn Schaerer. Robocup 2004 Presentation, 2004.

[13] M. Arbib. Schema Theory. In S. Shapiro, editor, *Encyclopedia of Artificial Intelligence*, pages 1427–1443. Wiley, 2nd edition, 1992.

[14] R. Arkin. Motor Schema-based Mobile Robot Navigation. *International Journal of Robotic Research*, 8(4):92–112, 1989.

[15] R. Arkin. Just what is a robot architecture anyway? Turing equivalency versus organizing principles. In *AAAI Spring Symposium: Lessons Learned from Implemented Software Architectures for Physical Agents*, 1995.

[16] R.C. Arkin. *Towards Cosmopolitan Robots: Intelligent Navigation in Extended Man-made Environments*. Ph.D thesis, University of Massachusetts, Department of Computer and Information Science, 1987.

[17] R.C. Arkin. Integrating Behavioral, Perceptual, and World Knowledge in Reactive Navigation. *Robotics and Autonomous Systems*, 6:105–122, 1990.

[18] R.C. Arkin. *Behavior-Based Robotics*. MIT Press, Cambridge, Massachusetts, USA, May 1998.

[19] Ronald C. Arkin and Tucker R. Balch. AuRA: Principles and Practice in Review. *Journal of Experimental and Theoretical Artificial Intelligence(JETAI)*, Volume 9(Number 2/3):175–188, April 1997.

[20] Ronald C. Arkin, Masahiro Fujita, Tsuyoshi Takagi, and Rika Hasegawa. Ethological modeling and architecture for an entertainment robot, 2001.

[21] Ronald C. Arkin and Douglas C. Mackenzie. Planning to behave: A hybrid deliberative/reactive robot control architecture for mobile manipulation, 1994.

[22] Jacky Baltes and John Anderson. Flexible binary space partitioning for robotic rescue. In *Proceedings of the IEEE/RSJ International Conference on Intelligent Robots and Systems (IROS)*, Las Vegas, October 2003.

[23] Jacky Baltes and John Anderson. Identifying robots through behavioral analysis. In *Proceedings of the Second International Conference on Computational Intelligence, Robotics, and Autonomous Systems (CIRAS)*, Singapore, 2003.

[24] Jacky Baltes and John Anderson. Interpolation Methods for Global Vision Systems. In Daniele Nardi, Martin Riedmiller, and Claude Sammut, editors, *The Seventh RoboCup Competitions and Conferences*, Berlin, 2005. Springer Verlag.

[25] Paul E. Black. *Algorithms and Theory of Computation Handbook*, chapter Dictionary of Algorithms and Data Structures. CRC Press LLC, 1999.

[26] R. P. Bonasso, R. J. Firby, E. Gat, David Kortenkamp, D. Miller, and M. Slack. Experiences with an architecture for intelligent, reactive agents. In *Journal of Experimental and Theoretical Artificial Intelligence*, volume 9, 1997.

[27] Michael E. Bratman, David J. Israel, and Martha E. Pollack. Plans and Resource-Bounded Practical Reasoning. *Computational Intelligence*, 4(4):349–355, 1988.

[28] Rodney A. Brooks. A robust layered control system for a mobile robot. In *IEEE Journal of Robotics and Automation*, volume 2, pages 14–23, 1986.

[29] Rodney A. Brooks. Intelligence without representation. Number 47 in Artificial Intelligence, pages 139–159. 1991.

[30] H. Van Brussel, R. Moreas, A. Zaatri, and M. Nuttin. A behaviour-based blackboard architecture for mobile robots. In *Industrial Electronics Society (IECON'98): 24th Annual Conference of the IEEE*, volume 4, pages 2162–2167, 1998.

[31] H. Burkhard, M. Hannebauer, J. Wendler, H. Myritz, G. Sander, and T. Meinert. BDI Design Principles and Cooperative Implementation — A Report on RoboCup Agents, 1999.

[32] Hans-Dieter Burkhard, Jan Wendler, Thomas Meinert, Helmut Myritz, and Gerd Sander. AT Humboldt in RoboCup-99. In *RoboCup*, pages 542–545, 1999.

[33] Chee Fon Chang, Aditya Ghose, Peter Harvey, and Justin Lipman. Gongeroos'99 Team. In *RoboCup-99 Team Descriptions*, 1999.

[34] Mark M. Chang, Brett Browning, and Gordon F. Wyeth. ViperRoos 2000. In *RoboCup 2000: Robot Soccer World Cup IV*, pages 527–530. Springer-Verlag Berlin Heidelberg, Inc., 2001.

[35] T. R. Collins, R. C. Arkin, and A. M. Henshaw. Integration of reactive navigation with a flexible parallel hardware architecture. In *IEEE Int. Conf. on Robotics and Automation*, volume 1, pages 271–276, 1993.

[36] J.H. Connell. SSS: A Hybrid Architecture Applied to Robot Navigation. In *1992 IEEE International Conference on Robotics and Automation*, volume 3, pages 2719–2724, May 1992.

[37] Ingo Dahm, Uwe Düffert, Jan Hoffmann, Matthias Jüngel, Martin Kallnik, Martin Lötzsch, Max Risler, Thomas Röfer, Max Stelzer, and Jens Ziegler. GermanTeam 2003. In *RoboCup 2003: Robot Soccer World Cup VII*, 2003.

[38] Raffaello D'Andrea, Tams Kalmr-Nagy, Pritam Ganguly, and Michael Babish. The Cornell Robocup Team. In *RoboCup 2000: Robot Soccer World Cup IV*, pages 41–51. Springer-Verlag Berlin Heidelberg, Inc., 2001.

[39] Vincent Decugis and Jacques Ferber. Action selection in an autonomous agent with a hierarchical distributed reactive planning architecture. In *Proceedings of the second international conference on Autonomous agents*, pages 354–361. ACM Press, 1998.

[40] R. G. Dromey. Architecture as an Emergent Property of Requirements Integration. In *Second International Workshop From SofTware Requirements to Architectures (STRAW'03)*, 2003.

[41] Marc Ebner. Evolution of a control architecture for a mobile robot. In Moshe Sipper, Daniel Mange, and Andrés Pérez-Uribe, editors, *Proceedings of the Second International Conference on Evolvable Systems: From Biology to Hardware (ICES 98)*, volume 1478, pages 303–310, Lausanne, Switzerland, 23-25 1998. Springer Verlag.

[42] RoboCup Federation. RoboCup website. http://www.robocup.org.

[43] FIRA. FIRA website. http://www.fira.net.

[44] R. James Firby. *Adaptive Execution in Dynamic Domains*. Ph.D thesis, Yale University, Jan. 1989.

[45] P. Furgale, J. Anderson, and J. Baltes. Real-Time Vision-Based Pattern Tracking Without Predefined Colors. In *Proceedings of the Third International Conference on Computational Intelligence, Robotics and Automation*, Singapore, December 2005. (to appear).

[46] Erann Gat. ALFA: A Language for Programming Robotics Control Systems. In *Proceedings of the IEEE Conference on Robotics and Automation*, May 1991.

[47] Erann Gat. Integrating planning and reacting in a heterogeneous asynchronous architecture for controlling real-world mobile robots. In *AAAI-92*, pages 809–815, July 1992.

[48] Andrey V. Gavrilov, Vasilij V. Gubarev, Kang-Hyun Jo, and H. H. Lee. Hybrid Neural-Based Control System for Mobile Robot. http://ermak.cs.nstu.ru/islab/publications/Kor2004.pdf.

[49] A. Goel, E. Stroulia, Z. Chen, and P. Rowland. Model-based reconfiguration of schema-based reactive control architectures. In *Proceedings of the Fifteenth International Joint Conference on Artificial Intelligence*. Morgan Kaufman Publishers, 1997.

[50] Tim Gorton and Bakhtiar Mikhak. A tangible architecture for creating modular, subsumption-based robot control systems. In *Extended abstracts of the 2004 conference on Human factors and computing systems*, pages 1469–1472. ACM Press, 2004.

[51] Ian Horswill. Functional programming of behavior-based systems. In *Autonomous Robots*, volume 9, pages 83–93. Kluwer Academic Publishers, Dordrecht, Netherlands, 2000.

[52] Andrew Howard. MuCows. In *RoboCup 2000: Robot Soccer World Cup IV*, pages 535–538. Springer-Verlag Berlin Heidelberg, Inc., 2001.

[53] H. Hu, J.M. Brady, J. Grothusen, F. Li, and P.J. Probert. LICAs: A Modular Architecture for Intelligent Control of Mobile Robots. In *Intelligent Robots and Systems 95. 'Human Robot Interaction and Cooperative Robots'*, volume 1, pages 471–476, 1995.

[54] Hui-Min Huang. An architecture and a methodology for intelligent control. In *IEEE Intelligent Systems and Their Applications*, volume 11, pages 46–55, April 1996.

[55] Trolltech Inc. Trolltech website. http://www.trolltech.com/.

[56] William Shotts Jr. linuxcommand.org website. http://linuxcommand.org/man_pages/xmllint1.html.

[57] R. M. Karp. Reducibility among combinatorial problems. In R. E. Miller and J.W. Thatcher, editors, *Complexity of Computer Computations*, pages 85–103, Plenum, New York, 1972.

[58] O. Khatib. Real-time obstacle avoidance for manipulators and mobile robots. *Int. J. Rob. Res.*, 5(1):90–98, 1986.

[59] Jong-Hwan Kim, Yong-Duk Kim, and Kang-Hee Lee. The Third Generations of Robotics: Ubiquitous Robot. In *Second International Conference on Autonomous Robots and Agents (ICARA)*, December 2004.

[60] Jong-Hwan Kim, Yong-Duk Kim, and Kang-Hee Lee. The Third Generations of Robotics: Ubiquitous Robot. In *6th IEEE International Symposium on Computational Intelligence in Robotics and Automation (CIRA)*, June 2005.

[61] Jong-Hwan Kim, Kang-Hee Lee, and Yong-Duk Kim. Ubiquitous Robot: The Third Generations of Robotics. In *Second American University of Sharjah Internation Symposium on Mechatronics (AUS-ISM)*, April 2005.

[62] Anne Koenig and Elisabeth Crochon. Tram: a blackboard architecture for autonomous robots. In *IEA/AIE '88: Proceedings of the first international conference on Industrial and engineering applications of artificial intelligence and expert systems*, pages 590–597, New York, NY, USA, 1988. ACM Press.

[63] Manuel Kolp, Paolo Giorgini, and John Mylopoulos. Organizational multi-agent architectures: A mobile robot example. In *AAMAS '02: Proceedings of the First International Joint Conference on Autonomous Agents and Multiagent Systems*, pages 94–95, New York, NY, USA, 2002. ACM Press.

[64] J.C. Latombe. *Robot Motion Planning.* Kluwer Academic Publishers, Boston, Massachusetts, USA, 1991.

[65] Tim Laue and Thomas Röfer. A behavior architecture for autonomous mobile robots based on potential fields. In *RoboCup 2004*, LNAI. springer, 2004.

[66] J. Ll de la Rosa, B. Innocenti, M. Montaner, A. Figueras, I. Munoz, and J. A. Ramon. RoGi Team Description. In *RoboCup 2001: Robot Soccer World Cup V*, pages 587–590. Springer-Verlag Berlin Heidelberg, Inc., 2002.

[67] Lotfi Zadeh. Fuzzy Sets. In *Information and Control*, volume 8, pages 338–353, 1965.

[68] Martin Lötzsch. XABSL: The Extensible Agent Behavior Specification Language. http://www.ki.informatik.hu-berlin.de/XABSL/.

[69] Martin Lötzsch, Joscha Bach, Hans-Dieter Burkhard, and Matthias Jüngel. Designing Agent Behavior with the Extensible Agent Behavior Specification Lan-

guage XABSL. In *RoboCup 2003: Robot Soccer World Cup VII*, pages 114–124. Springer-Verlag Berlin Heidelberg, Inc., 2003.

[70] Kian Hsiang Low, Wee Kheng Leow, and Jr. Marcelo H. Ang. A hybrid mobile robot architecture with integrated planning and control. In *Proceedings of the first international joint conference on Autonomous agents and multiagent systems*, pages 219–226. ACM Press, 2002.

[71] P. Maes. How to do the Right Thing. *Connection Science Journal*, 1(3):291–323, February 1989.

[72] P. Maes. Situated Agents can have Goals. *Robotics and Autonomous Systems*, 6:49–70, 1990.

[73] F. Michaud, G. Lachiver, and C.T. Le Dinh. A New Control Architecture Combining Reactivity, Planning, Deliberation and Motivation for Situated Autonomous Agent. In *Fourth International Conference on Simulation of Adaptive Behavior*, pages 245–254, 1996.

[74] David P. Miller, Rajiv S. Desai, Erann Gat, Robert Ivlev, and John Loch. Reactive navigation through rough terrain: experimental results. In *Proceedings Tenth National Conference on Artificial Intelligence - AAAI-92*, pages 823–828, Jul 1992.

[75] Monica N. Nicolescu and Maja J. Matarić. A hierarchical architecture for behavior-based robots. In *Proceedings of the first international joint conference on Autonomous agents and multiagent systems*, pages 227–233. ACM Press, 2002.

[76] Fabric Noreils and Raja Chatila. Plan execution monitoring and control architecture for mobile robots. *IEEE Transactions on Robotics and Automations*, 2, 1995.

[77] R. Peter Bonasso and David Kortenkamp. An intelligent agent architecture in which to pursue robot learning. In *Working Notes: MCL-COLT '94 Robot Learning Workshop*, July 1994.

[78] S. Quinlan and O. Khatib. Experimental robotics 2. In R. Chatila and G. Hirzinger, editors, *Towards real-time execution of motion tasks*. Springer-Verlag, 1993.

[79] Ananth Ranganathan and Sven Koenig. A Reactive Robot Architecture with Planning on Demand. In *International Conference on Intelligent Robots and Systems(IROS)*, pages 1462–1463, 2003. Las Vegas.

[80] R. John Reh. Pareto's Principle - the 80-20 rule. http://management.about.com/cs/generalmanagement/a/Pareto081202.htm.

[81] Thomas Röfer, R. Brunn, Ingo Dahm, M. Hebbel, Jan Hoffmann, Matthias Jüngel, Tim Laue, Martin Lötzsch, W. Nistico, and M. Spranger. GermanTeam 2004. In *RoboCup 2004: Robot Soccer World Cup VIII*, 2004.

[82] Raúl Rojas, Sven Behnke, Lars Knipping, and Bernhard Frötschl. FU-Fighters 2000. In *RoboCup 2000: Robot Soccer World Cup IV*, pages 547–550. Springer-Verlag Berlin Heidelberg, Inc., 2001.

[83] Raúl Rojas, Sven Behnke, Achim Liers, and Lars Knipping. FU-Fighters 2001 (Global Vision). In *RoboCup 2001: Robot Soccer World Cup V*, pages 571–574. Springer-Verlag Berlin Heidelberg, Inc., 2002.

[84] Stuart Russell and Peter Norvig. *Artificial Intelligence: A Modern Approach*. Prentice Hall, 1995.

[85] G.N. Saridis. Architectures for intelligent controls. *Intelligent Control Systems: Theory and Applications*, 1995.

[86] Reid Simmons. An architecture for coordinating planning sensing and action. In *Proceedins of the Workshop on Innovative Approaches to Planning, Scheduling, and Control*, 1990.

[87] Marc G. Slack. Sequencing formally defined reactions for robotic activity: Integrating raps and gapps. In *Proceedings of the SPIE's Conference on Sensor Fusion*, 1992.

[88] Monnett Hanvey Soldo. Reactive and Preplanned Control in a Mobile Robot. In *1990 IEEE International Conference on Robotics and Automation*, volume 2, pages 1128–1132, May 1990.

[89] Alexander Stoytchev and Ronald C. Arkin. Combining deliberation, reactivity, and motivation in the context of a behavior-based robot architecture, 2000.

[90] AliReza Fadaie Tehrani, Peyman Amini, Hamid Reza Moballegh, Pezhman Foroughi, Omid Teheri, Behrouz Touri, Ahmad Movahedian, and Mohammd

Ajoodanian. IUT Flash Team Description. In *RoboCup 2003: Robot Soccer World Cup VII*, 2003.

[91] Jason Thomas, Kenichi Yoshimura, and Andrew Peel. Roobots. In *RoboCup 2001: Robot Soccer World Cup V*, pages 591–594. Springer-Verlag Berlin Heidelberg, Inc., 2002.

[92] Daniel Toal, Colin Flanagan Caimin Jones, and Bob Strunz. Subsumption architecture for the control of robots. In *IMC-13*, pages 703–711, 1996.

[93] A. M. Turing. Computing Machinery and Intelligence. *Mind*, 49(236):433–460, 1950. http://cogprints.org/499/00/turing.html.

[94] T. Tyrrell. An evaluation of Maes's bottom-up mechanism for action selection. *Adaptive Behavior*, 2(4), 1994.

[95] Eric W. Weisstein. Traveling Salesman Problem. http://mathworld.wolfram.com/TravelingSalesmanProblem.html.

[96] Alfredo Weitzenfeld, Ronald C. Arkin, Francisco Cervantes, Roberto Olivares, and Fernando Corbacho. A Neural Schema Architecture for Autonomous Robots. In *International Symposium on Robotics and Automation*, Saltillo, Coahuila, Mexico, Dec. 12–14 1998. http://www.cc.gatech.edu/ai/robot-lab/online-publications/Iberamia.pdf.

[97] Mattias Werner, Helmut Myritz, Uwe Düffert, Martin Lötzsch, and Hans-Dieter Burkhard. Humboldt Heroes. In *RoboCup 2000: Robot Soccer World Cup IV*, pages 651–654. Springer-Verlag Berlin Heidelberg, Inc., 2001.

[98] Wikipedia. NP-Hard. http://en.wikipedia.org/wiki/NP-hard.

[99] World Wide Web Consortium (W3C). Extensible Markup Language (XML). http://www.w3.org/XML/.