Intelligent and Robust Control of Autonomous Mobile Robots

Gerald Steinbauer

Dissertation

vorgelegt zur Erlangung des akademischen Grades
Doktor der Technischen Wissenschaften

an der Technischen Universität Graz
Graz, Oktober 2006

Abstract

This thesis presents a framework for the intelligent robust control of autonomous mobile robots which carry out various tasks in real-world environments. The framework has been incrementally developed for the deployment in two major application areas, the RoboCup robot soccer and the service robotics domain.

The proposed framework is flexible in order to be applicable in different domains and for different tasks. This flexibility is reached by module-based design of the software, an open organization of the functions of the framework, and by the use of paradigms from the distributed computing community. Furthermore, the framework comprises a strong deliberative component which due to the planning and reasoning capabilities enables the controlled robot to carry out complex tasks. A general task description language enables an easy and intuitive specification of a wide range of different tasks. Moreover, the framework is able to robustly execute a tasks in dynamic and unpredictable real-world environments and under the presence of noise, of uncertainty in perception and execution and of faults at runtime in the robot system. This robustness is achieved by a guarded plan execution, a robust mapping from the quantitative to the qualitative representation of the world and a model-based diagnosis and repair system.

Finally, a evaluation of the framework in successful experiments in the real-world is presented and shortcomings of the proposed framework and future research directions are discussed.

Acknowledgement

Doing a PhD thesis in the domain of autonomous mobile robots and especially in RoboCup Middle-Size League is never an one-man show. There is a whole bunch of people which helped and supported me during the work for this thesis.

First of all I like to thank my supervisor Franz Wotawa. He set a lot of faith in me and gave me the freedom to set up a complete RoboCup Middle-Size League team and to develop and realize a lot of my ideas. Furthermore, he gave me a brilliant scientific guidance in order to learn doing research. Moreover, he highly supported me with numerous fruitful discussions about my work, with his wisdom about how a university works, a lot of funding for new robots, sensors and equipment and in the jointly establishing of the field of autonomous mobile robots at the Graz University of Technology.

I also like to thank the second supervisor Uwe Egly from University of Technology Vienna for his feedback to this thesis.

My great thanks goes to all current and previous members of the *Mostly Harmless* RoboCup Middle-Size Team and particularly to Arndt Mühlenfeld, Gordon Fraser, Roland Koholka, Jürgen Wolf, Stefan Galler, Martin Weiglhofer, Mathias Brandstötter, Michael Hammer, Gerald Krammer, Martin Buchleitner, Simon Jantscher, Jörg Weber and Christine Wagner. Without their never ending patience many of the ideas and successes of the team and this work never have been possible.

Furthermore, I like to thank all the researchers, professors, the deans, the rectors, the administrative staff, the people from the workshops and all other employees of the Graz University of Technology which supported our RoboCup Team and me during the last years.

My love and thank goes to my parents, my brothers and my new family who taught me to never stop being curious, supported me during the endless days of tournament preparation and always encouraged me to learn and try new things.

Last but not least I am grateful to Petra Pichler the “secret head” of the Institute for Software Technology. She kept away from me almost everything of the administrative and financial nightmares to give me the time and mood to do more funny things like research and playing with robots.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

In this chapter we will discuss the motivation and the background which stand behind this thesis and the work carried out during its preparation. We will show how science and application meet in the area of autonomous mobile robot. Moreover, we will describe two popular application areas of autonomous mobile robots (robot soccer and service robots) in detail, which are the basic application areas of the work in this thesis. The two application scenarios will guide us throughout the reminder of this thesis. Afterwards, we discuss open questions which arise from the above two scenarios. Finally, we briefly sketch our proposed solutions of the problems and our contributions to the field of autonomous mobile robots.

1.1 Background and Motivation

Autonomous mobile robots have gained an increased attention by the research community during the last decade. On the one hand the robots themselves raise a lot of scientific question but on the other hand, mobile robot have reached a status where they can serve as a real-world testbed for various research like machine learning, evolutionary algorithms, economy and biology inspired methods. But also in our society the awareness of such kind of robots increase. In the meanwhile almost everybody is aware of or definitely knows a mobile robot system. Today, there are a lot of different robots deployed in competitions like the RoboCup, floor and window cleaning, logistic domains, of course military applications and many other domains. The good news is that most of these mobile robots have reached robustness and public acceptance in these application areas.

It can be foreseen that in the near future the demand of mobile robots which are able to autonomously carry out various task will significantly increase. Therefore, the range of the character of different tasks a robot is able to perform and the operational environments will dramatically increase. Furthermore, due to the expected number of unexperienced user which will have first contact with such a robot the demand of autonomy and robustness of the deployed

systems also will further increase.

From the research perspective the demand on autonomous mobile robots is very interesting. Due to the variety of the robots, the tasks and the environments a lot of questions and challenges emerge. For us one question is of major interest. How can we develop control systems for autonomous mobile robots which are able to robustly control a robot in different tasks in dynamic environments under the presence of noise, uncertain sensor data and faults?

In the next section we describe two popular scenarios which are used in the area of research on autonomous mobile robots in order to answer the above question.

1.1.1 RoboCup Robotic Soccer

The Robot World Cup Initiative (RoboCup for short) is an international attempt by universities and research centers to foster Artificial Intelligence and intelligent robotics research by providing a standardized problem that poses a tough challenge for several scientific disciplines and technologies. The first RoboCup competition was held 1997 at IJCAI in Nagoya. The interest in RoboCup and the number of participating teams have increased every year since then. In 2005 the RoboCup was held in Osaka Japan and attracted about 2000 participants in 330 teams from 31 nations [BJNT06]. Until a team of robots is actually able to perform a soccer game, various technologies have to be incorporated, including multi-agent cooperation, strategy acquisition, real-time reasoning, machine learning, robotics, perception, vision and sensor-fusion. Contrary to other autonomous mobile robots, which are optimized for a single heavy-duty task, robot soccer is a task for a team of cooperative fast-moving robots in a fast changing environment.

To interest and educate young students and researchers in the field of AI and Robotics is also an important goal of the RoboCup. Throughout every year the RoboCup Federation organizes a number of national and international competitions, conferences and workshops. These events are great opportunities to objectively evaluate your work during a competition, to present and discuss your approaches and new ideas.

The RoboCup is organized in several leagues. They differ in several aspects: simulated or real robots, the types of sensors (global or local), and the size and type of the robots. Hence, RoboCup provides the optimal platform and testbed for various research topics.

Part of the RoboCup is the so called *Middle Size League* (MSL). Our team *Mostly Harmless* [1] [SBB+06] participates in this league since 2003 and we use this league as a testbed for our research. In the MSL, teams of up to six robots whit a approximately size of 50 cm times 50 cm times 80 cm compete. Figure 1.1 shows a game situation in an international tournament. The size of the field is currently up to 14m x 12m. The major difference to other leagues or

[1] The name Mostly Harmless originates from the title of the 5^{th} book of the famous book series "The Hitchhiker's Guide to the Galaxy" by Douglas Adams.

competitions, in addition to the size of the robots and the field, is that no global or external sensor systems for perception are allowed. Thus the robots have to rely totally on their own sensors, including vision. The robots are fully autonomous, i.e., their sensors, actuators, power supply and computational power are on-board, and no external intervention by humans is allowed, except to insert robots to or remove robots from the field. External computational power is allowed, but most teams use it only for monitoring purposes. Wireless communication between the robots and/or with the external computer is also allowed. As in most of the other RoboCup leagues, relevant objects are distinguishable by their color: the ball is orange, the goals are yellow and blue, the robots are black, the field lines are white, the robot markings (to distinguish the teams) are magenta and light blue. The middle-size league provides a serious challenge for research disciplines such as cooperative multi-robot teams, autonomous navigation, sensor fusion, vision-based perception, planning, reasoning and mechanical design, to name only a few of them.

Figure 1.1: Two robots of the RoboCup MSL Team of the Graz University of Technology (left) during a game at the RoboCup German Open 2004.

In the past years the community decided to remove the surroundings of the field for the middle-size league (2002 the walls were replace by a group of poles, 2003 the poles were also removed), as a step towards a more realistic soccer game and to post new scientific challenges. As the ball and also the robots are able to leave the field now and the surrounding scenery is not defined anymore, the demands on the robots regarding ball handling, perception, and strategy increase. It could also be foreseen that in the near future the field size further will be enlarged

in order to foster cooperative play and that the field will be outside which raises a lot of new challenges for the perception, e.g., robust computer vision.
Currently Artificial Intelligence for planning and cooperation do not yet play such an important role for the success in the tournament as, for example in other leagues which use simulated agents. But for a long-term scientific development these are the relevant issues. The problems caused by perception (mainly vision), self-localization, mechanical and electronic design (ball handling, robot drives and sensors) are still dominating, and make it difficult to implement adaptive, intelligent, and cooperative team play. All this control issues are the interesting question we try to answer with our proposed approach of intelligent robust control.

1.1.2 Service Robots

The ultimate goal of research in the area of service robots is to relieve people from hard, dangerous and monotonous work. The dream of having a mechanical companion which does all the boring work for us is as old as the mankind.

Figure 1.2 shows the prototype of our outdoor service robot which is based on a Pioneer AT3. Its task is to deliver good like books or letters within our university campus. The campus is about 1000 m times 500 m large and comprises all kind of buildings, streets, parking lots and parks. The requirements for the control system of this robot are similar to those in the previous section. But the quality and the intensity of the problems are much higher than in the regulated soccer domain. The operational time is much longer and the environment is less restricted but more dynamic due to open spaces, pedestrians, cars and so forth. Furthermore, the environmental conditions like ambient light vary much more which increases the noise of the perception. Fortunately, it is possible to transfer many of the approaches for the robot control from the soccer domain to the service robot domain. But some of these approaches have to be scaled or adapted for the increased demands.

We use the service robots domain as the second scenario for the research on robust intelligent control of autonomous mobile robots. The use of this domain has three major aspects. Firstly, it is the natural extension of the regulated testbed of RoboCup. Secondly, many techniques from RoboCup can be transfered to and can be evaluated in this different domain. Moreover, the service domain post additional challenges for the developed methods and for robotics research in general. Finally, the service robot domain will have a lot of impact on the economy and the society in the near future. On one hand the industry demands for intelligent logistic system and on the other hand we face with the problems of an ageing society. Therefore, the demand for intelligent autonomous robots will significantly increase.

Figure 1.2: Outdoor Service Robot.

1.2 Problem Statement

This thesis is focused on is abstract intelligent robust control of autonomous mobile robots. As said previously this kind of robot will gain even more attention in the near future and their application for non-trivial tasks in general “everyday live” environments will significantly increase. These scenarios raise a lot of scientific questions in order to enable robots to act robust and really autonomous for various tasks and in different environments. Such a robot has to be equipped with an intelligent robust control framework. Such a feasible framework should fulfill the following requirements:

- **Flexibility and reuse of components:** Core components of the framework should be reusable and a new arrangement of the components within a new framework should be easily possible. Furthermore, the framework and its components should enable the robot to perform a wide range of different tasks without significant modifications or even recompilation.

- **Planning and reasoning for complex tasks:** Complex task cannot be carried out by a robot without the capability of reasoning and planning. Therefore, an appropriate deliberative component have to be part of th control system. Such capabilities furthermore demand for an appropriate abstract logic-based representation of the knowledge of the robot.

- **General, expressive and intuitive task description:** A robot and its control should be as flexible as possible in order to carry out many different tasks. Therefore, a task description language is needed which is general and expressive enough to describe a wide range of different task in different worlds. Furthermore, such a task description should be intuitive in order to allow also non-experienced users to specify a task for the robot.

- **Robust task execution in noisy and dynamic environments:** Robots usually carry out their tasks in the real world. The disadvantage of the real world is that it is inherently noisy, uncertain and dynamic. Therefore, the perception of the robot and outcome of actions the robot performs are uncertain to a certain level. Furthermore, the world is dynamic and may evolve in a way the robot has not foreseen. Therefore, the robot has to have the capability to robustly execute task in the presence of noise, dynamic and exogenous events.

- **Fault-tolerance:** Faults in the software and hardware of mobile robots are inherent. A robot which autonomously performs a task has to have some level of fault-tolerance. It is desirable that the robot is able to detect and localize faults in his system and is able to set the appropriate repair or control actions in order to be able to complete its mission or at least to proceed to a safe mode.

The aim of this thesis is to integrate as many as possible of the above features in a general framework for the intelligent robust control of autonomous mobile robots. The next section list the contributions we have made to the scientific community in order to develop our framework.

1.3 Major Contribution

The scientific contribution of this thesis is the incremental development of a framework which allows a flexible and robust control of an autonomous mobile robot in different real-world application like the RoboCup Middle-Size League and the service robotic domain. The developed framework solves different problems which arise from the dynamic behavior and uncertainty of the real world and the complexity of non-trivial tasks for a group of autonomous mobile robots.

- **Software framework:** The first significant contribution achieved during the preparation of this thesis was the development of an open flexible software framework for the control of autonomous mobile robots. This framework serves as the base for all further research done during this thesis. The further research mainly concerned the robust intelligent control of mobile robots. Therefore, much work has been done in the area of hybrid control architectures of mobile robots. The developed framework is based on the Miro-framework of the University of Ulm [USEK02]. Many of our developments find their way back to

the original framework and are used by several international research groups. All further developments were integrated into the framework in order to achieve the goal of a intelligent robust control of autonomous mobile robots. The framework was described in [FSW04b] and was used in different application domains like RoboCup robot soccer and service robotics.

- **Prediction of movement:**

  We have developed a novel prediction method for the movement of objects. The approach is used to predict the movement of the ball in sequences of camera images in the RoboCup environment. The method is based on Machine Learning and the recently proposed computational paradigm of the Liquid State Machine. The Liquid State Machine comprise a heavily interconnected pool of spiking neurons (*the liquid*) and a relatively simply set of readout neurons. The liquid projects the input data in a high-dimensional space were simpler readout methods like linear regression can be used. The visual input is presented to a visual receptor field and the appropriate prediction was trained. The main advantage is that arbitrary non-linear predictions can be performed by the approach. The work was published in [BKLS05] and received a nomination for the Best Paper Award at the 18^{th} International Conference on Industrial & Engineering Applications of Artificial Intelligence & Expert Systems (IEA/AIE) in 2005.

- **Robust plan execution:**

  As shortly motivated above, only a hybrid control architecture (a combination of reactive and deliberative control) seems to be appropriate to robustly control a autonomous mobile robot carrying out a complex task. Therefore, an abstract top level of control with reasoning and planning capabilities has to be part of the architecture. Usually, planning modules lack of reactivity. This is a major drawback in highly dynamic domains like robot soccer. We have developed a general robust framework for planning and plan execution in mobile robots.
  The main contribution in this area was the development of *plan invariants*. The invariants are part of each planning problem and are permanently monitored during plan execution. Plan invariants enables a quick reaction to exogenous effects which invalid a long-term plan. Such invalidation may be not recognized or even later without the use of the invariants. Moreover, the parts of the planning problem for a given task easily can be exchanged for different applications or robot capabilities. This exchangeability forms a sort of deliberative programming language which can be used in different domains. The planning framework is successfully used in several different application areas like robot soccer and service robotics and is described in [FSW05].

- **Robust symbol grounding:**

  Another drawback of the use of deliberative control is that in any serious robot system at some point in the system one has to switch from the quantitative representation of the world to a qualitative abstract knowledge representation. This process is known in the literature as symbol grounding [RN03].
  Due to the quality and the noise of the robots sensors, the quantitative world model is inherently effected by uncertainty. We like to avoid uncertainty or probabilities on the qualitative level because the complexity of the knowledge representation and the planning and reasoning process will drastically increase. Due to the uncertain nature of the quantitative representation we inherit problems like predicate oscillation on the qualitative level. In order to make this symbol grounding more robust, we propose a new mechanism for the calculation of the truth value of predicates. The approach is based on the well known hysteresis mechanism. The idea is to delay the change of the truth value until a real significant change in the quantitative world takes place. The new symbol grounding was applied and evaluated in the RoboCup domain. The approach and the results were published in [FSW04a] and [SWW05].

- **Model-based diagnosis for mobile robots**

  Even if the hardware and software of an autonomous mobile robot is carefully designed, implemented and tested, there is always the possibility of a fault at runtime. But some level of redundancy and robustness against such fault is crucial for a truly autonomous robot. In order to improve the robustness of the control system against faults at runtime, we enriched the software framework with fault detection and localization capabilities.
  The diagnosis capabilities are based on model-based diagnosis. The used correct behavioral model of the system is mainly derived from the communication means of the different software components of the system. Due to the CORBA-based architecture of the control software, the derivation of the model and the monitoring of the communication is quite simple. Deviations of the desired and the actual behavior of the system is detected by a set of observers. The output of the observers triggers the diagnosis and is used in the fault localization by logical reasoning. Moreover, we developed an approach which allows the system to correct detected faults on the fly.
  The diagnosis system was described in [SW05b] and [SW05c] and successfully tested and evaluated in the RoboCup domain [SMW05]. We received for the proposed approach the *RoboCup Engineering Challenge Award* at the RoboCup 2005 in Osaka. Furthermore, we worked on the integration of diagnosis of software and hardware of mobile robots. The different nature of the both domains raised a lot of new challenges in the domain of model-based diagnosis [SW05a].

1.4 Related Research

Prior research related to the topics of this thesis, alternatives or former approaches are presented and discussed within the corresponding chapter.

1.5 Structure of the Thesis

The remainder of the thesis is organized as follows. In the following chapter the developed software framework running on our robots is motivated and described. Furthermore, the requirements for such software frameworks and other existing frameworks will be discussed in more details. Moreover the developed hardware of our RoboCup soccer robots will briefly described. In Chapter 3 a novel method for the prediction of object movements in series of images is presented. The prediction is based on the liquid state machine. Their computational paradigm enables a prediction of complex movements of objects. The next chapter covers intelligent robust control. In this chapter we introduce and describe mechanisms for robust deliberative control of robots in dynamic and partially unpredictable domains. Chapter 5 discuss the problem of bridging the quantitative and the qualitative representations in a world of uncertain sensors and perception. Moreover a solution to the problems in deliberative control caused by this uncertainty is presented. The following chapter describes a framework for Model-Based Diagnosis for the control software of autonomous mobile robots. Furthermore, the chapter shows how the framework is extended towards automated repair of faults at runtime. The Chapter 7 discuss how diagnosis of hardware and software can be integrated in a framework which performs automated diagnosis and repair of faults in a mobile robot platform. The following chapter raises some directions for future research. Finally, Chapter 9 presents a summary of the work and draws some conclusions.

Chapter 2

Frameworks for Mobile Robots

2.1 The need for an appropriate framework

In research in the area of autonomous mobile robots an appropriate software framework is crucial. The framework should allow to robustly control different mobile robots during the execution of a wide range of different tasks. Moreover, it should be flexible enough to allow to investigate different control strategies and algorithms. Furthermore, the extension of the framework towards a handling of more complex tasks and environment should be easily possible. Finding the appropriate framework for a special purpose is a very challenging task. Moreover, no one will expect to develop a single framework which is appropriate for all purposes. There is always the tradeoff between general applicability and usability. The question for an appropriate framework can be divided into two parts. The first and more easy to answer part is which control paradigm is used. The second part is more related to engineering and concerns the appropriate software architecture. In [Ore04] the topic of the choice and implementing of an appropriate framework is discussed in more detail. Parts of this chapter were published in [FSW04b].

2.1.1 Robot Control Paradigm

The robot control paradigm guides the organization of the control of a mobile robot which enables the robot to perform given tasks. It structures how the robot maps its sensor readings to actions via a more or less intelligent decision making module. One of the first attempts to structure control was the *Sense-Plan-Act* (SPA) paradigm. Figure 2.1 depicts the paradigm. The paradigm was inspired by the research on Artificial Intelligence of the late 60's and was first successfully used by Nilsson in the robot *Shakey* [Nil84]. It was guided by the early view on Artificial Intelligence. The paradigm divides the control into three functionalities. *SENS* is responsible for the perception of the robots internal state and its environment. The data provided by

the robot's sensors are interpreted and combined to a central abstract model of the world. Based on the information in the world model, a description of the capabilities of the robot and the goal of the task the *PLAN* module tries to find a plan (i.e., a sequence of actions) which will lead to a given goal. Such a planning problem comprises an initial state I, a set of possible actions A and the desired goal state G. The *ACT* module executes this plan in order to achieve the goal. Although, the SPA paradigm is very powerful and flexible it suffers from a set of drawbacks. First of all, planning needs a lot of time even on very powerful computers. Therefore, the reaction to dynamic environments is slow. Planning algorithms generally work on a qualitative and abstract representation of the world. The design of such a representation and the transformation of quantitative sensor data into this representation are far from being trivial.

The reactive *Sense-Act* (SA) control paradigm in contrast provides a completely different organization of control. Figure 2.2 depicts the SA paradigm. The paradigm is biologically inspired by the mechanism of reflexes which directly couples the sensor input with the actor output. Such a reflex of the robot is commonly called a behavior. More complex behaviors emerge trough the combination of a set of different reflexes. A system which follows this paradigm was first proposed in the mid 80's with the Subsumption Architecture by Brooks [Bro86]. This architecture brought a big progress in the research on mobile robots and is still popular and widely used. Brooks argued that abstract knowledge about the world and reasoning is not necessary for the control of a mobile robot. The paradigm is able to control a robot also in a dynamic environment because the reaction time is very slow due to the encoding of the desired behavior into a reflex and the tight coupling of the sensors and actors. Although, relatively complex behaviors can be achieved by blending different reflexes, the paradigm is prone to fail for more complex tasks. This arises from the fact that no explicit information about the internal state of the robot and about the world and no additional knowledge about the task is used. Therefore, for complex tasks a goal-driven approach is much more appropriate than a simple instinct-driven one.

Although, the choice of an appropriate control paradigm sometimes seems to be more a question of faith than science, there is a relatively clear commitment within the robotics research community that the most appropriate architecture is a hybrid architecture (see Figure 2.3). Hybrid systems combine the advantages of the planning and the reactive paradigm while avoiding most of their drawbacks. Such systems use reactive behaviors where reactivity is needed (e.g., avoiding a dynamical obstacle) and use planning and reasoning when complex decisions have to be performed and some delay is not critical. Usually such systems comprise three layers. The *Reactive Layer* uses reactive behaviors to implement a fast coupling of the sensors and the actors in order to be reactive to a dynamic environment. Very often this layer implements the basis skills of a robot like e.g. basic movement primitives and obstacle avoidance. The *Deliberative Layer* has a global view on the robot and its environment and is responsible for high-level planning in order to achieve a given goal. Typical functionalities which are located in this layer

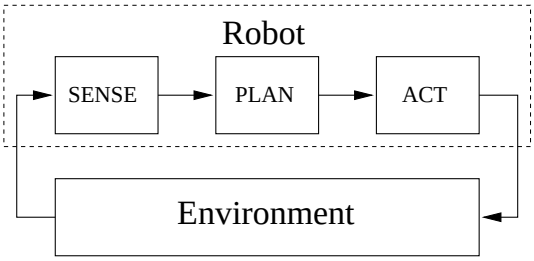

Figure 2.1: The *Sense-Plan-Act* control paradigm.

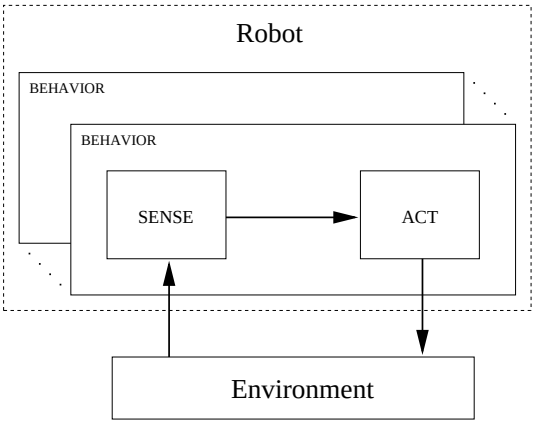

Figure 2.2: The reactive *Sense-Act* control paradigm.

are mission planning, reasoning, localization, path planning and the interaction with humans or other robots. The *Sequence Layer* is located between the Reactive and the Deliberative Layer and bridges the different representation of the two layers. The sequencer generates a set of behaviors in order to achieve a subgoal submitted by the Deliberative Layer. It is also responsible for the correct execution of such a set of behaviors and should inform the higher layer if the subgoal was successfully reached or the execution failed due to some reason.

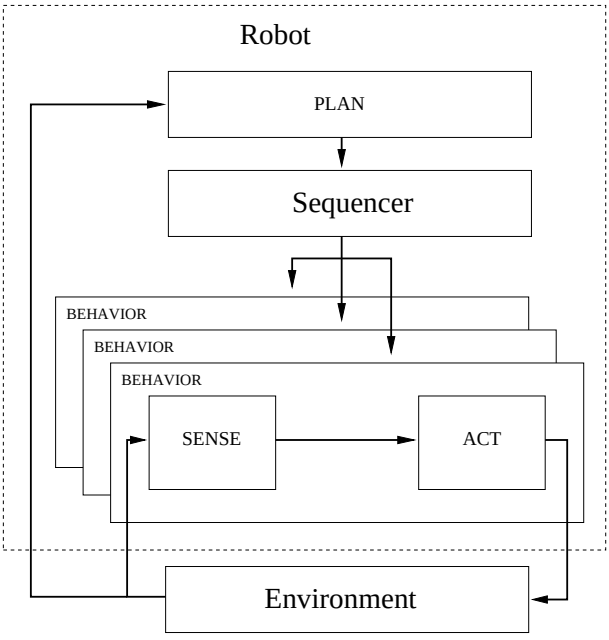

Figure 2.3: The hybrid control paradigm.

A more deep introduction into the different control paradigm can be found in the book by Kortenkamp and colleagues [KBM98] and the book by Murphy [Mur02].

2.1.2 Software Architecture

The control paradigm guides the functional decomposition of the control on a more abstract view. The software architecture on the other hand guides the decomposition in different components and the implementation of such components. Furthermore, it concerns about the encapsulation of different functionalities into manageable modules. A good software architecture should also provide among others the following features:

- robustness
- flexibility
- sensor and actor interface abstraction
- easy exchange and reuse of components
- reliable communication between components
- easy adaptation of the system for new purposes
- easy porting to other hardware platforms
- support of a defined development process
- support for test and evaluation

The question about the software architecture is not tightly coupled to pure robotic research. Therefore, the above requirements and principles are ignored in most of the implementations of prototypic robot research software. Consequently, most of the research software is hard to maintain, to port and to extent and therefore lacks of general usability. But fortunately, many of the best-practice principles and processes from the software development community are now widely accepted by the robotic research community. These principles among others are object-orientated design, the use of design patterns, the reuse of established libraries, the use of widely accepted standards and the use of test and evaluation frameworks. This leads to higher quality, more exchangeable and more flexible software. Furthermore, a great pool of software frameworks for robotic research have been created. Most of them can be used out of the shelf and fulfil most of the requirements proposed by robotic research. Some of these frameworks will be discussed more deeply in the next section.

2.2 Existing frameworks for mobile robots

This section introduces some popular existing frameworks for mobile robot research. Some of them are more general and flexible than others while some of them are closely related to specific robots or tasks. The advantages and the drawbacks of the different frameworks will be presented. A very good overview and a more formal and detailed evaluation of existing frameworks is given in [OC03].

2.2.1 Task Control Architecture (TCA)

The Task Control Architecture (TCA) was developed by Reid Simmons at the Carnegie Mellon University [Sim94].

TCA allows you to construct a distributed system without having to build your own remote procedure call mechanism. At its core, TCA provides a flexible mechanism for passing messages between processes (which were called modules). The communication mechanisms automatically marshal and unmarshal data, invoke user-defined handlers when a message is received, and include both publish/subscribe and client/server type messages, and both blocking and non-blocking types of messages. TCA also provides orderly access to robot resources so that you don't have to build your own queuing mechanism. This features are also now available separately from TCA in the *Inter Process Communication* (IPC) library.

TCA simplifies building task-level control systems for mobile robots. By "task-level", TCA means the integration and coordination of perception, planning and real-time control to achieve a given set of goals (tasks). TCA provides a general control framework, and it is intended to be used to control a wide variety of robots. TCA provides a high-level, machine independent method for passing messages between distributed machines. Although TCA has no built-in control functions for particular robots (such as path planning algorithms), it provides control functions, such as task decomposition, monitoring, and resource management that are common to many mobile robot applications. The development of high-level control is supported by the *Task Description Language* (TDL) [SA98]. The language consists statements to handle task and task control. The control program written in TDL is transformed into C++ code, which will be compiled and linked to the TCA core.

TCA can be thought of as a robot operating system — providing a shell for building specific robot control systems. Like any good operating system, the architecture provides communication with other tasks and the outside world, facilities for constructing new behaviors from more primitive ones, and means to control and schedule tasks and to handle the allocation of resources. At the same time, it imposes relatively few constraints on the overall control flow and data flow in any particular system. This enables TCA to be used for a wide variety of robots, tasks, and environments. One successful example is the robot XAVIER [SGH+97].

2.2.2 Saphira

Saphira has been developed by Kurt Konolige at the Stanford Research Institute (SRI) International [KM98]. It is an integrated architecture for robot perception and control and was first developed within the Flakey robot project. Saphira is a well known and widely used framework because it is shipped as the basic software suite with the commercial research robots of Pioneer family by ActivMedia. Therefore, Saphira is now tightly coupled to these robots.

Saphira comprises of two architectural layers. The *System Architecture* provides basic communication and interfaces to the actors and sensors of a robot and is implemented by the *ARIA* library. The interface to higher layers in the hierarchy are provided by the *state reflector*, a container for an abstract view of the internal state of the actors and sensors of a robot. ARIA is maintained by ActivMedia and provides access only to the hardware of robots of the Pioneer family. The second layer is the *Control Architecture* and is build on top of the state reflector. The Control Architecture comprises modules for controlling a robot. These modules mainly concern navigation. The main features within this layer are the *Local Perception Space* (LPS), which contains a robot-centric view on the environment up to a few meters radius around the robot, and the *Global Map Space* (GMS), which provides a global view on the environment and its structure. Furthermore, this layer provides the possibility to easily set up reactive behaviors using a fuzzy blending of behaviors. The implementation of higher level tasks is supported by the C-like scripting language *Colbert* [Kon97]. Colbert contains statements for a wide range of control concepts. A big advantage of Saphira is that advanced methods for obstacle avoidance, localization and path planning are integrated and ready-to-use.

Although Saphira is quite common in robotic research and provides an easy start for the work with robots of the Pioneer family, it has some drawbacks which lower the value for general use. Saphira has been evolved over a long period of time. Therefore, the design and implementation is somehow awful. A porting of Saphira to other robot platforms is nearly impossible. Finally, an adaptation for specific tasks beyond the functionality provided by the Colbert language is very exhaustive.

2.2.3 Carnegie Mellon Robot Navigation Toolkit (Carmen)

The *Carnegie Mellon Robot Navigation Toolkit* (Carmen) is an open-source collection of software for mobile robot control developed at the Carnegie Mellon University [MRT03]. Carmen is modular software designed to provide basic navigation primitives including actor and sensor control, obstacle avoidance, localization, path planning, people-tracking and mapping.

Carmen was designed to provide a consistent interface and a basic set of primitives for robotics research on a wide variety of commercial robot platforms. The ultimate goals of Carmen are to lower the barrier to implementing new algorithms on real and simulated robots and

to facilitate sharing of research and algorithms between different institutions. Robotics research covers a spectrum of different approaches and formalisms. The developers have adopted the philosophy of making Carmen as inclusive as possible.

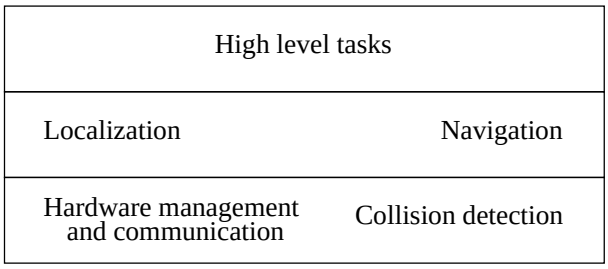

Figure 2.4: The layered architecture of Carmen.

Carmen uses a three-tier architecture (see Figure 2.4). The *base layer* provides abstract interfaces to sensors and robot platforms. There exist already a wide range of implementations of the interfaces for robot research platforms, e.g., the Pioneer family from ActivMedia, the B21 and the the ATRV family from iRobot, and for different range sensors, e.g. the Sick LMS 200. Unfortunately, Carmen currently only provides interfaces and implementations for robot platforms equipped with a differential drive. Furthermore, the base layer provides simple control loops, e.g., motion primitives. The *navigation layer* implements navigation primitives, e.g., localization, path-planning and object tracking. Carmen comprises a large set of ready-to-use implementation of advanced algorithms for path-planning [Kon00] and for laser and map-based localization [FBDT99]. There also exist abstract interfaces which enable an easy extension of Carmen with new navigation capabilities. The last layer hosts the user-level tasks which are based on the functionality of the lower layers. All functionalities across the layers are encapsulated in small modules with clear interfaces. Carmen uses the IPC library (see Section 2.2.1) for the communication between those modules.

Finally, Carmen provides excellent visualization tools and tools for automated mapping of different environments. Although Carmen provides an easy begin for the development of a robot control software and a wide range of off-the-shelf components, the use of Carmen is mostly related to the localization and navigation research of the developers.

2.2.4 Open Robot Control Software/Open Realtime Control Services (OROCOS)

Open Robot Control Software [Bru01] and *Open Realtime Control Services* [BSK03] are the two parts of the OROCOS open-source project. The project aims the ambitious goal of providing

both general standards and designs for robot control applications and ready-to-use implementations of modules guided by these standards and designs. The project's main feature is that it provides interchangeable open-source implementation developed under state-of-the-art software development principles. Because well established software development principles are hardly found in todays robot control software.

While other existing frameworks share the common goals of interchangeability, reuse and common interfaces with OROCOS, the project goes one step further. It also tries to establish basic standards of notations for e.g. coordination systems, kinematic and motion control only to name a few. These feature will also engage the interchange of methods and code between the robot research community. The system has a very open and extensible design enabling an easy contribution of methods and codes to the project. Another advantage of the project is that it supports hard real-time control which very much widens the possible application areas for the projects.

The project provides already a wide range of control modules ranging from a simple PID motor controller to the complete control of a six-DOF robot arm. Mobile robots are currently not covered by the project. These types of robots are in the focus of future extensions to the project.

2.2.5 Player/Stage

Player is a device server for sensors and actors of a mobile robot [GVS+01]. The development has been carried out as a open source project by a number of people which contribute to the project. It provides a connection to the sensors and actors of a robot for a client through simple TCP/IP sockets.

The message format is standardized for different types of sensors and actors. Therefore, some level of hardware abstraction is provided. While the use of a communication by simple socket is slim and efficient the work with messages comprising of a chunk of bytes is somehow cumbersome. But one design goal of Player is to be very efficient to be able to serve a near unlimited number of clients at the same time. However, the use of sockets as communication mechanism provides independence of the used OS and programming language. The modular design of the Player server enables an easy integration of new hardware into the server. Player uses a uniform abstraction for various devices by the UNIX-like treating of devices as files. Reading data from a sensor is done by an ordinary read on its device node and sending commands to an actor is done by an ordinary write to its device node. The server already supports a number of commercial research robots like the Pioneer family by ActivMedia and the B21 by iRobot.

While Player is an efficient interface to sensors and actors and is used by many researchers it has a main drawback. It neither provides mechanisms or modules for reactive control of a robot nor it provides any deliberative layer. Therefore, up from the sensor and actor level all parts of

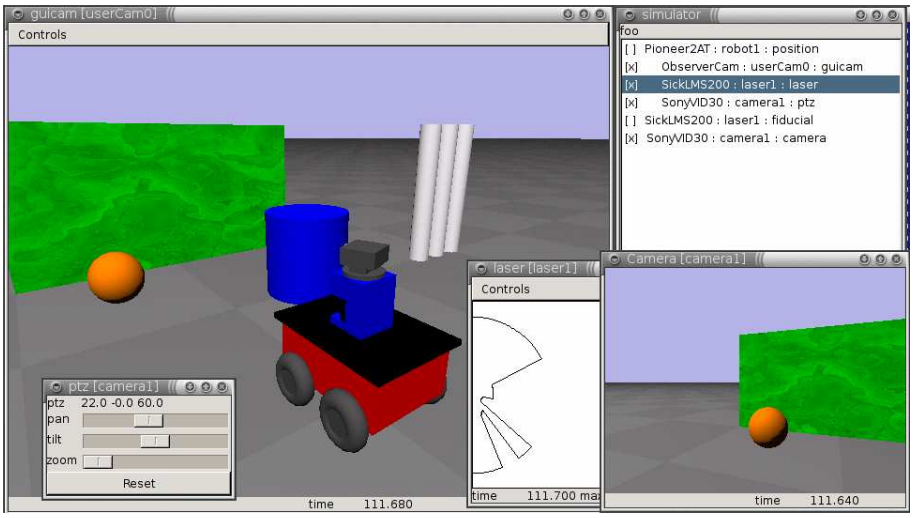

Figure 2.5: Screen shot of a simulation done with Gazebo.

the robot control have to be implemented by the user.

Together with Player comes *Stage*. Stage is a simulator for a group of robots in a two-dimensional bit-mapped environment. Stage simply can be used as plug-in to the Player device server and simulates the behavior of the actors and sensor in the given virtual environment. Stage is able to simulate a wide range of different robot platforms and a great number of sensors including sonar, laser scanner and odometry. Due to the usage of the *Open Dynamics Engine* (ODE) a very realistic simulation encountering many phenomena, e.g. collisions and acceleration, is provided. Recently, with Gazebo a Player-compatible simulator for realistic three-dimensional environments was introduced into the project. Figure 2.5 shows a screen shot of Gazebo simulating a Pioneer 3AT robot equipped with a laser-scanner and a camera.

2.2.6 Middleware for Robots (Miro)

The *Middleware for Robots* (MIRO) is a distributed object oriented software framework for robot applications. It has been developed at the Department of Computer Science at the University of Ulm [USEK02, Utz05]. The aim of the project is to provide an open flexible software framework for applications on mobile robots. The goals for the design of Miro comprise the following:

- full object-oriented design
- client/server System design
- hardware and operating system abstraction

- open architecture approach
- multi-platform support, communication support and interoperability
- software design patterns
- agent technology support

Miro achieved these goals by an architecture which is divided into three layers. Figure 2.6 depicts the architecture of Miro. The usage of the *Adaptive Communication Environment* (ACE) and CORBA for the communication between the layers and other applications enable a flexible, transparent and platform-independent development. Miro uses *The ACE Object Request Broker* (TAO) as CORBA framework. The implementation of the framework is completely performed object-oriented in the C++ programming language.

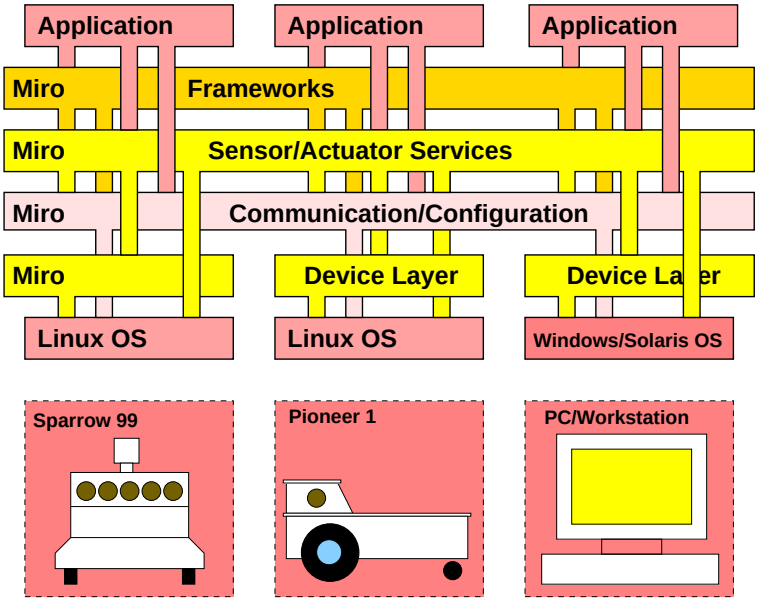

Figure 2.6: The architecture of Miro [Utz05].

The *Miro Device Layer* provides object-oriented interface abstractions for all sensory and actuatory facilities of a robot. This is the platform-dependent part of Miro. The *Miro Communication and Service Layer* provides active service abstractions for sensors and actuators via CORBA *Interface Definition Language* (IDL) descriptions and implements these services as network-transparent objects in a platform-independent manner. The programmer uses standard CORBA object protocols to interface to any device, either on the local or the remote robot. The *Miro Framework* provides a number of often used functional modules for mobile robot control, like

modules for mapping, self-localization, behavior generation, path planning, logging and visualization facilities.

Although Miro is a software framework and no ready-to-use robotic application, it was decided to use Miro as the basis for our own control software. A complete description of Miro and many useful examples can be found in [Tea05]. The use of Miro as a basis for further developments has the following advantages:

- **Object oriented design**: The design of the framework is fully object-oriented, elaborated and easy to understand. Moreover, there are a whole bunch of ready-to-use design patterns like, e.g., multi-threading, device reactors and so forth.

- **Multi-Platform Support and Reuse**: Miro comprises a great number of abstract interfaces for numerous different sensors and actors, e.g., odometry, bumper, sonar, laser and differential-drives. Moreover, for all of these interfaces, already implementations for many different robot platforms are provided. Due to the clear design and the use of CORBA and IDL the implementation of interfaces for a new robot platform and the integration of new interfaces is straight forward. Miro currently supports many different common robot platforms like the B21 and the Pioneer family. Furthermore, many research groups use Miro for the control of their robots. We like to mention here the RoboCup Middle-Size Teams of the University of Ulm [KMU+04], Technical University of Munich [SKGB04] and Graz, University of Technology [SBB+06]. Recently, contributions of different research groups have been integrated into Miro. These contributions mainly concern interfaces and service for new hardware, e.g., GPS receiver, compasses and robot platforms.

- **Communication**: For the communication between different components of the robot control software Miro, provides two main mechanism.

  Direct CORBA method calls are used in a client/server manner for a one-to-one communication of components. These mechanism is usually used for actor and sensor interfaces. Due to the use of CORBA, the user has not to deal with the internals of such a communication, e.g., marshalling or memory management. Furthermore, the communication is completely transparent even if the client and the server run on different computers or use different programming languages.

  The event channel on the other hand provides on-to-n communication. The event channel follows the producer/consumer paradigm. The producer simply pushes an event of a certain type to the channel. All consumers which are subscribed for this event are automatically informed if this specific event is available. Although this mechanism has a lot of advantages, it has to be mentioned that a heavy use of this mechanism leads to a poor run-time performance due to the computational overhead in the event channel.

Another extremely useful communication mechanism is the notify multicast. Roughly spoken it is a very reliable Event Channel for inter-robot communication. The usual CORBA communication mechanism like the remote method invocation and the event channel use TCP/IP network connections in case of communication between different robots or computers. Such communication is unreliable and performs very bad in environments with a bad quality of and a high traffic on its network connections. An example for such environments is the wireless network situation at RoboCup tournaments, where a high number of clients communicates with high traffic over larger distances. In these environment frequently the network connection totally collapses. The notify multicast instead uses UDP packages distributed over multicast groups. It provides the same functionality as the common event channel like e.g. offer and subscription of events but uses a slimmer and more reliable transport mechanism. Furthermore, groups of heterogeneous robots are able to communicate. By using a general description language for data, like e.g. positions, states and objects, teams of autonomous mobile robots were able to successfully play with a mixed team at the RoboCup 2004. The mixed teams comprised robots from the RoboCup Middle-Size teams of Ulm, Munich and Graz [USM05].

- **Behavior Engine**: Miro contains a complete module for the modeling and the implementation of reactive behaviors. The *Behavior Engine* follows the behavioral control paradigm introduced by Brooks [Bro86]. The module uses a hierarchical decomposition of behaviors. On the base of the hierarchy there are different simple *behaviors* like e.g. wall following. These behaviors can be grouped in *action patterns*. Such action patterns may comprise e.g. a wall following and a local obstacle avoidance behavior. Different action patterns can be combined to a *policy*.

  The transition between different action patterns are triggered by two different mechanisms. The local transitions are emitted by a behavior and have a unique name. An action pattern is linked by this name to a successor action pattern which will be activated next. Global transitions directly contain the action pattern which will be activated next. Within an activated action pattern, all its behaviors are executed concurrently. Each action pattern has an arbiter which combines the output of the concurrent behaviors and communicates the output to the actuators. Currently only a prioritized arbiter is available in Miro. But one can think about of more advanced arbiters which provide a more smooth transition between behaviors.

  The type of a behavior is one of the following three types. For the *TimedBehaviors* type the behavior is called repeatly at fixed timesteps. The behavior itself has to assure that its calculation is finished within the timestep. Otherwise it would block the other behaviors. *EventBehaviors* are called every time one of the subscribed events occur. The *TaskBehav-*

iors type is used for behaviors which may not be able to finish their calculations within a fixed time. Such behaviors run within their own task while not blocking other behaviors.

Once the behaviors are implemented action patterns and policies are build up by describing them in a XML-file. Therefore experiments with different action patterns and policies are easy and straight forward.

Unfortunately, Miro do not provide any paradigms and implementations for a deliberative layer. Therefore, the Miro Framework was extended by our own planning system. Details about this and other contributions by our group are described more detailed in the next section.

2.3 The Developed Framework

In order to fulfill as many as possible of the above requirements for an appropriate framework, a novel design approach for mobile robots has been developed [FSW04b]. The approach is based on a continuous modularization of both the robot's software and its hardware. The hardware modularization is based on an encapsulation of the robot's various physical skills into autonomous modules with defined interfaces. Therefore, hardware modules can be exchanged very simply. This allows an easy adaptation of the robot's hardware for new tasks and simple investigation of new modules or new module configurations. The modularization of the software is based on two concepts: (1) software design and (2) software architecture. The software design provides a decomposition of functionality into layers with increasing levels of abstraction. The design is inspired by the hybrid control paradigm. Therefore, the functionality is organized in different layers ranging from an abstract top layer with planning and reasoning capabilities down to a layer with direct hardware access. The software-architecture which is based on Middleware for Cooperative Robotics (Miro) [USEK02] deals with the implementation details. The Miro framework provides several ready to use interfaces to sensors and actors, methods for an integration of new software modules into the framework and reliable transparent communication mechanisms between software modules. The software modules are implemented as autonomous services which interact via client/server communication. Due to the object-oriented design and the existence of defined interfaces, the adaptation of the framework to our platform was quite easy. Furthermore, based on this fact, an exchange of software modules within the community is possible. E.g., the framework was extended by adding interfaces to new hardware (e.g., Firewire, CAN-Bus). These extensions are now publicly available. This design approach was used for the development of the robots that form our RoboCup Middle-Size League Team (MSL) [SFF+03]. As a result the team was able to create a robot soccer team from scratch with limited human and financial resources within less than a year. Besides using the robots for soccer games, the robots

are used in research in the area of service robots.

2.4 Hardware Design

In our previous studies, four skills were found which are important for a mobile robot in order to fulfill a given task in a given environment, regardless whether the robot plays soccer or delivers mail within an office building. These skills are: (1) movement, (2) sensing the environment, (3) manipulating the environment and (4) information processing. These skills may differ from task to task, e.g., a kicking mechanism in a robot soccer tournament or a manipulator arm with a gripper in the service robot domain. An encapsulation of these skills in different loosely coupled modules is the first step to a flexible hardware design. Therefore, the hardware of the robot is divided into four layers. Each layer provides one of those skills. The layers are stacked to build up the robot platform (see Figure 2.7).

There are no restrictions to the design of the layers themselves except that they have to provide the required skills and three predefined interfaces. A mechanical interface ensures that individual layers fit together mechanically. The fast reliable Can-Bus allows the communication within the layers [Ets01]; each layer is able to communicate directly with each other layer. A single 24 V power line provides the power supply for the layers. The Can-Bus and the power line are simply looped through the layers. Introducing new layers that provide different characteristics of a skill is easily possible by following the guidelines introduced above. Every layer is equipped with its own processing unit, either a C167 microcontroller or a Pentium-based Single-Board PC. Therefore, the individual layers are able to work autonomously.

2.4.1 Driving Layer

The Driving Layer is responsible for handling the movement of the robot. In our current design this layer is implemented as an omnidirectional drive. It is built up by four orthogonal crosswise motors each joint to an omni-wheel. By individual control of the speed and the rotating direction of each motor the robot is able to move in any direction and to rotate around its vertical axis simultaneously. This layer also hosts the battery packs to keep the center of gravity of the robot low.

2.4.2 Actuator Layer

All active interactions with the environment are done by the Actuator Layer. This layer is implemented as a pneumatic kicking device for the purpose of playing soccer games or a manipulator arm with a gripper during service tasks.

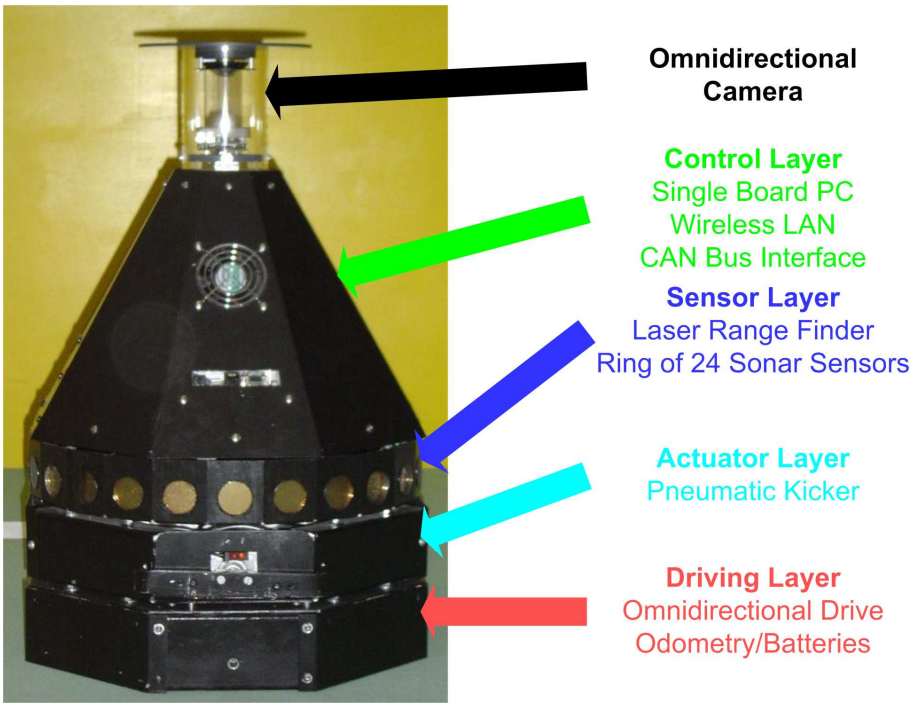

Figure 2.7: The modularized robot platform.

2.4.3 Sensor Layer

The Sensor Layer provides the entire sensing of the environment. This layer actually hosts two sensor systems. We use a Sick Laser Range Finder for proximity scans around the robot. This sensor has a high resolution (0.5°) and provides very reliable measurements. A disadvantage of this sensor is the limitation of the scan to 180° around the robot. Therefore, this sensor is supported by another sensor system, a ring of 24 ultrasonic sensors. Those sensors have a significantly lower resolution and accuracy compared to the laser scanner but provide a qualitative scan around the entire robot.

2.4.4 Control Layer

The more sophisticated information processing is done in the Control Layer. This includes more advanced sensory data processing, the decision making process and higher level control. Therefore, this layer is equipped with a powerful processing unit, an 850 MHz Pentium III Single Board PC with 256 MB Ram and a 20 GB hard drive. This layer also provides a communication channel to other robots or computers via a Wireless-LAN interface. Due to the field of view and the connection via the Firewire-Interface the omnidirectional camera is mounted on top of

this layer. A crucial constraint for the assembly of this layer is the use of standard interfaces (e.g., PCI, USB, Firewire, CAN-Bus) instead of proprietary ones. This eases the exchange of components within this layer.

2.5 Software Design

The design of the software is guided by a continuous modularization. This modularization is divided into two important aspects of the design. The first aspect deals with the functional organization of the software. It introduces a decomposition of the software in parts of similar functionality and an abstraction into layers. The second aspect deals with the logical organization of the software modules and the communication within these modules. Whereas the first aspect is important for the design and understanding of the behavior of the robot in a more abstract way, the second aspect is important for the software implementation. This distinction eases the development process due to the fact that the designer of the behavior does not have to deal with software implementation aspects and vice versa. The idea of functional layers with different levels of abstraction is similar to the idea of cognitive robotics [CGI+02]. As mentioned above a combination of reactive behaviors, explicit knowledge representation, planning and reasoning capabilities promises to be more flexible and robust. Furthermore, such an approach will be able to perform far more complex tasks. The software design is in fact inspired by the hybrid control paradigm. But it has to be mentioned that the proposed design differs from the general hybrid paradigm that our design has no sequence layer. The tasks of the sequence layer are located together with the reactive behaviors in one layer. The functionality of the software is divided into three layers with an increasing level of abstraction. The functionality of a layer is based on functionality of the layer below. The layers are shown in Figure 2.8.

2.5.1 Hardware Layer

The Hardware Layer implements the interfaces to the sensors and actuators of the robot. This layer delivers raw continuous sensory data and performs a lowlevel controlling of the actuators. USB for the Laser Range Finder and Firewire for the omnidirectional camera are standard interfaces and already supported by our OS (Linux). The interfaces to modules on the CAN-Bus are implemented as Virtual *CAN-Connections*. A software module is able to transparently communicate via these connections directly with one dedicated hardware module on the CAN-Bus. The method is similar to the well known TCP/IP protocol where different applications on different computers are able to communicate over one physical network connection.

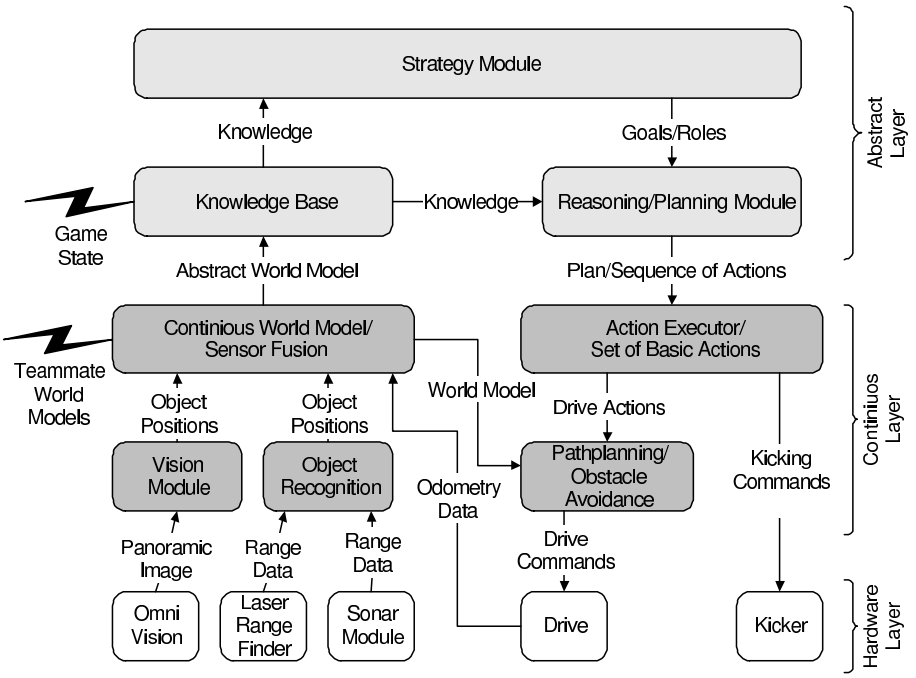

Figure 2.8: Functional view of the software (robot soccer example).

2.5.2 Continuous Layer

The Continuous Layer implements a numerical representation (quantitative view of the world) of the sensing and acting of the robot. This layer performs the processing of range data and the image processing. This processing provides possible positions of objects in the environment including the robots own pose. A pose consists of position and orientation of an object. The pose together with the motion information from the odometry are fused into a continuous world model. The sensor fusion is done by Kalman Filters (object positions) [DGN01] and Monte Carlo methods (own pose) [FBDT99]. For sure, all sensing and acting of a real mobile robot is afflicted with uncertainty. Therefore, sensor fusion is done using the above probabilistic methods. The world model represents the continuous world by estimating the most likely hypothesis for the positions of objects and the position of the robot itself. Furthermore, this layer is responsible for the execution of actions. Execution is based on a set of actions implemented as patterns of prioritized simple reactive behaviors.

2.5.3 Abstract Layer

The Abstract Layer implements a symbolic representation (qualitative view of the world) about the knowledge of the robot and a planning module for the decision making. A detailed description

of the used planning system can be found in [Fra03]. A similar approach also has been proven to work in the RoboCup MSL domain [DFL02]. The core of this layer is the Knowledge Base. It contains the entire higher-level knowledge of the robot. This knowledge consists of previously collected domain knowledge, an abstracted representation of the continuous world model and an abstract description of the actions the robot is able to perform. This knowledge is represented using a STRIPS-like representation language [FN71] enriched by the use of elements of first-order logic (e.g., quantifiers). Based on this knowledge, the strategy module chooses the next goal the robot has to achieve for fulfilling the longterm task. The Planing Module generates a plan (sequence of basic actions) which satisfies this goal. This plan is monitored permanently for its validity during execution. The plan is canceled or updated if preconditions or invariants of the plan or its actions are no longer valid. This plan is communicated to the Action Executor which performs the actions of the plan. The Abstract Layer allows for an easy implementation of a desired task by specifying the goals, actions and knowledge as logic sentences.

2.6 Software Architecture

The Software Architecture is based on Miro [USEK02]. The Software Architecture is shown in Figure 2.9. All software modules are implemented as autonomous services. Each service runs as an independent task. The communication between services is primarily based on two mechanisms: (1) CORBA-Interfaces and (2) event channel. CORBA-Interfaces are described using IDL and export methods a service is able to perform. The IDL description of the interface is abstract and makes no assumptions about the implementation of the interface, e.g, programming language or platform. The use of IDL provides abstraction of the sensor and actor interfaces. The event channel is a mechanism which collects and delivers events within the system. It enables one-n communication. A service that produces an event simply pushes the event in the Event Channel. A service which consumes an event simply subscribes to some event type on the Event Channel. If some event of that type is available, the Event Channel delivers the event to the subscribed service. The advantage of the Event Channel is that producers and consumers do not have to be aware of each other in contrast to CORBA-Interfaces, where the client has to know the server in advance. Hence, the services are independent and an adaptation of software modules or the integration of new services is very easy and transparent. Based on the flexible design of the framework, some extensions to the framework have been developed. These extensions are mainly interfaces to new hardware used in our robots like the Firewire interface for digital cameras and the CAN-Bus interface with virtual connections. Clearly, these extensions are provided to the public. These extensions widen the number of platforms on which the framework could be used.

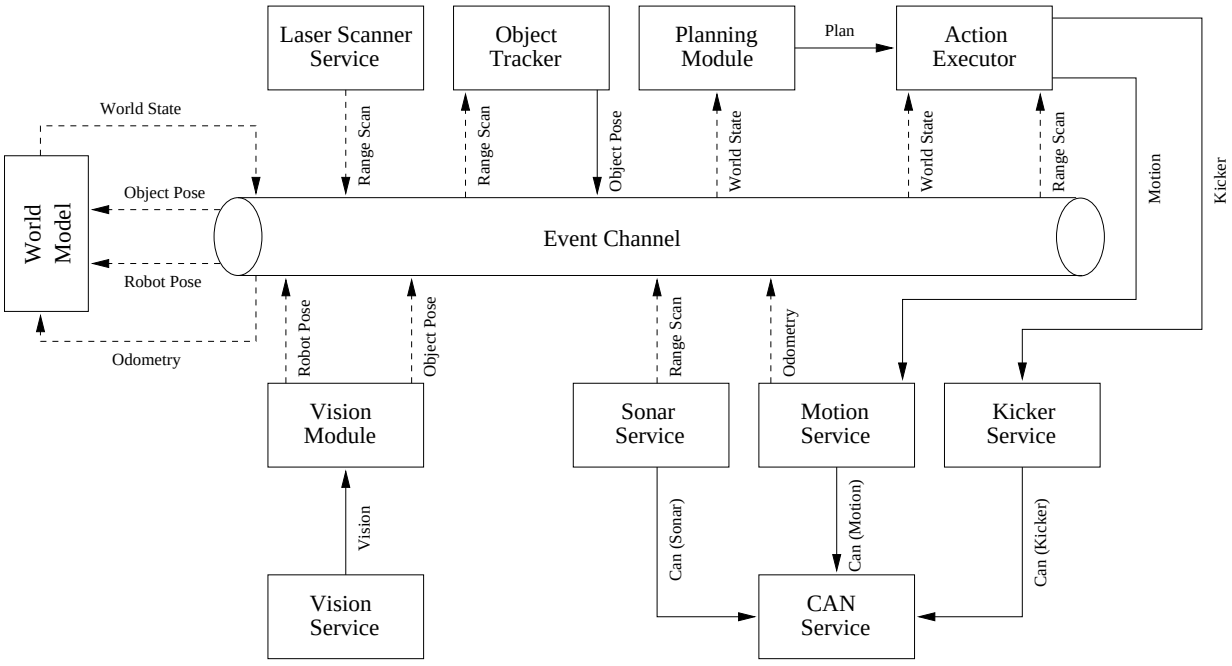

Figure 2.9: Software Architecture. Solid connections represent CORBA IDL interfaces. Dashed connections represent communication via an event.

2.7 Obtained Results and Discussion

The experiences in building a mobile robot show that this kind of design approach reduces the time and costs for developing a mobile robot and at the same time increases the flexibility and robustness of the robot. By using the design approach our group was able to develop a robot soccer team from scratch with limited human and financial resources within less than a year. The basic system was implemented by a team of only nine students (32 man-months). Another practical result shows the advantages of the modularized hardware design. Two versions of the Sensor Layer based on different ultrasonic sensors (Polaroid or Devantec) were developed. Both versions work transparently within the robots. The quality and robustness of our robots were shown during a number of RoboCup tournaments where the hardware and software of the robots ran stable. Moreover, the adaptability and flexibility of our control solution were impressive during the tournaments. A player's behavior could be changed easily in a few minutes on the field simply by modifying sentences that to some extent resemble human language statements. Furthermore, the framework is nearly unchanged used for successful controlling a delivery robot within our institute.

Although the used framework has proved to be flexible and robust by the use in different domains, it suffers from two major drawbacks: (1) communication delay on the event channel

and (2) the time needed for planning. The extensively use of the event channel as a flexible communication mechanism slows down the system and reduces the reactivity of the robot in dynamic environments. Due to use of an abstract decision making module with AI planning the robot is able to deal with very complex tasks. There are many decisions which are made in the Abstract Layer. Therefore, the system is even less reactive because planning needs much time. In the future, a better balance between the deliberative and the reactive layer has to be achieved.

In this chapter, an approach for designing autonomous mobile robots has been presented. The approach is based on a continuous modularization in software and hardware. Robot designs that are based on our approach are more flexible with regard to their intended purpose and can be easily adapted to new tasks. The hardware modularization is based on an encapsulation of skills. The software modularization is based on the framework Miro. Due to the CORBA-based architecture of the software modules an adaptation of available services and the integration of new services is very easy. The behavior of the robot itself can be easily adapted due to the different abstraction layers in the functionality and a logic based representation of knowledge, goals and actions.

Chapter 3

Looking ahead

3.1 Introduction

The prediction of time series is an important issue in many different domains, such as finance, economy, object tracking, state estimation and robotics. The aim of such predictions could be to estimate the stock exchange price for the next day or the position of an object in the next camera frame based on current and past observations. In the domain of robot control such predictions are used to stabilize a robot controller. See [JW99] for a survey of different approaches in motor control where prediction enhances the stability of a controller. In this chapter we present a novel approach for prediction in the robotics domain. This chapter was partially published in [BKLS05].

There are two popular approaches for this kind of prediction: (1) modeling the behavior of the system or (2) learning of the prediction based on collected data. The former approach claims a basic understanding of the underlying system. It is preferred if the internal structure of the system is well know and its behavior could be sufficiently precise described by a set of equations, i.e., electronic circuits, technical processes or mechanical systems. A well known example for this approach is the prediction step in state estimation with the Kalman-Filters [May90]. It uses the current state and a linear system model to predict the state for the next time step. This prediction is optimal for linear systems. For non-linear systems, the Extended Kalman-Filter (EKF) uses a linearization of the system. Therefore, the EKF is not optimal anymore. The latter approach is to learn the prediction from previous collected data. The advantages are that knowledge of the internal structure is not necessarily needed, arbitrary non-linear prediction could be learned and in addition some past observations could be integrated in the prediction.

Artificial Neural Networks (ANN) are a common method used for this computation. *Feed-forward networks* only have connections starting from external input nodes, possibly via one or more intermediate hidden node processing layers, to output nodes. *Recurrent networks* may have

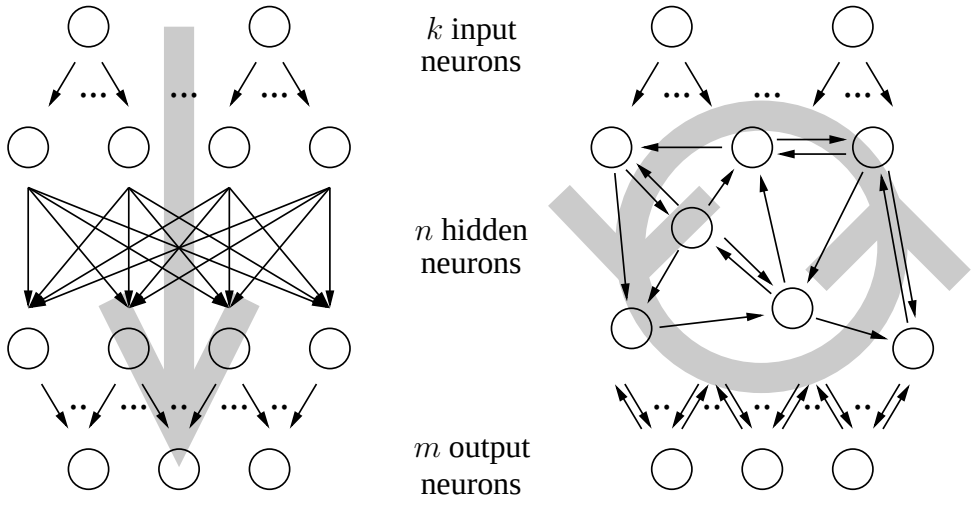

Figure 3.1: Comparison of the architecture of a feed-forward (left hand side) with a recurrent neural network (right hand side); the grey arrows sketch the direction of computation.

connections feeding back to earlier layers or may have lateral connections (i.e. to neighboring neurons on the same layer). See Figure 3.1 for a comparison of the direction of computation between a feed-forward and a recurrent neural network. With this recurrence, activity can be retained by the network over time. This provides a sort of memory within the network, enabling it to compute functions that are more complex than just simple reactive input-output mappings. This is a very important feature for networks that will be used for computation of time series, because a current output is not solely a function of the current sensory input, but a function of the current and previous sensor inputs and also of the current and previous internal network states. This allows a system to incorporate a much richer range of dynamic behaviors. Many approaches have been elaborated on recurrent ANNs. Some of them are dynamic recurrent neural networks [Pea95], radial basis function networks [Bis95], Elman networks [Elm90], self-organizing maps [Koh01], Hopfield nets [Hop82] and the “echo state” approach from [Jae01].

In case of autonomous agents, it is rather difficult to employ strictly supervised learning algorithms for recurrent ANNs such as back-propagation, Boltzmann machines or Learning Vector Quantization (LVQ), because the correct output is not always available or computable. It is also very difficult to set the weights of a recurrent ANN directly for a given non-trivial task. Hence, other learning techniques have to be developed for ANN that could simplify the learning process of complex tasks for autonomous robots. The liquid state machine – which will be introduced in the next subsection – is one approach that overcomes these difficulties.

Recently, networks with models of biologically more realistic neurons, e.g., spiking neurons, in combination with simple learning algorithms have been proposed as general powerful tools

for the computation on time series [MNM02]. In Maass et. al. [MLM02] this new computation paradigm, a so called *Liquid State Machine* (LSM), was used to predict the motion of objects in visual inputs. The visual input was presented to a 8x8 sensor array and the prediction of the activation of these sensors representing the position of objects for succeeding time steps was learned. This approach appears promising, as the computation of such prediction tasks is assumed to be similar in the human brain [Bea00]. The weakness of the experiments in [MLM02] is that they were only conducted on artificially generated data. The question is how the approach performs with real-world data. Real data, e.g. the detected motion of an object in a video stream from a camera mounted on a moving robot, are noisy and afflicted with outliers.

In this chapter we present how this approach can be extended to a real world task. We applied the proposed approach to the RoboCup robotic-soccer domain. The task was movement prediction for a ball in the video stream of the robot's camera. Such a prediction is important for reliable tracking of the ball and for decision making during a game. The remainder of this chapter is organized as follows. The next section provides an overview of the LSM. Section 3.3 describes the prediction approach for real data. Experimental results will be reported in Section 3.4. Finally, in Section 3.5 we draw some conclusions.

3.2 The Liquid State Machine

The LSM from [MNM02] is a new framework for computations in neural micro-circuits. The term "liquid state" refers to the idea to view the result of a computation of a neural micro-circuit not as a stable state like an attractor that is reached. Instead, a neural micro-circuit is used as an *online computation tool* that receives a continuous input that drives the state of the neural micro-circuit. The result of a computation is again a continuous output generated by readout neurons given the current state of the neural micro-circuit.

Recurrent neural networks with spiking neurons represent a non-linear dynamical system with a high-dimensional internal state, which is driven by the input. The internal state vector $x(t)$ is given as the contributions of all neurons within the LSM to the membrane potential of a readout neuron at the time t. The complete internal state is determined by the current input and all past inputs that the network has seen so far. Hence, a history of (recent) inputs is preserved in such a network and can be used for computation of the current output. The basic idea behind solving tasks with a LSM is that one does *not* try to set the weights of the connections within the pool of neurons but instead reduces learning to setting the weights of the readout neurons. This reduces learning drastically and much simpler supervised learning algorithms which e.g., only have to minimize the mean square error in relation to a desired output can be applied.

The LSM has several interesting features in comparison to other approaches with recurrent circuits of spiking neural networks:

1. The liquid state machine provides “any-time” computing, i.e., one does not have to wait for a computation to finish before the result is available. Results start emitting from the readout neurons as soon as input is fed into the liquid. Furthermore, different computations can overlap in time. That is, new input can be fed into the liquid and perturb it while the readout still gives answers to past input streams.

2. A single neural micro-circuit can not only be used to compute a special output function via the readout neurons. Because the LSM only serves as a pool for dynamic recurrent computation, one can use many different readout neurons to extract information for several tasks in parallel. So a sort of “multi-tasking” can be incorporated.

3. In most cases simple learning algorithms can be used to set the weights of the readout neurons. The idea is similar to support vector machines, where one uses a function (usually called kernel) to project input data into a high-dimensional space. In this very high-dimensional space simpler classifiers can be used to separate the data than in the original input data space. The LSM has a similar effect as a kernel. Due to the recurrence, the input data is also projected to a high-dimensional space. Hence, in almost any case experienced so far, simple learning rules like, e.g., linear regression are sufficient.

4. Last but not least it is not only a computational powerful model, but it is also one of the biological most plausible so far. Thus, it provides a hypothesis for computation in biological neural systems.

The model of a neural micro-circuit as it is used in the LSM is based on evidence found in [GWM00] and [TWWB02]. Still, it gives only a rough approximation to a real neural micro-circuit since many parameters are still unknown. The neural micro-circuit is the biggest computational element within the LSM, although multiple neural micro-circuits could be placed within a single virtual model. In a model of a neural micro-circuit $N = n_x \cdot n_y \cdot n_z$ neurons are placed on a regular grid in 3D space. The number of neurons along the x, y and z axis, n_x, n_y and n_z respectively, can be chosen freely. One also specifies a factor to determine how many of the N neurons should be inhibitory. Another important parameter in the definition of a neural micro-circuit is the parameter λ. Number and range of the connections between the N neurons within the LSM are determined by this parameter λ. The probability of a connection between two neurons i and j is given by

$$p_{(i,j)} = C \cdot exp^{-\frac{D_{(i,j)}}{\lambda^2}} \tag{3.1}$$

where $D_{(i,j)}$ is the Euclidean distance between those two neurons and C is a parameter depending on the type (excitatory or inhibitory) of each of the two connecting neurons. There exist

four possible values for C for each connection within a neural micro-circuit: C_{EE}, C_{EI}, C_{IE} and C_{II}. The subscripts are used depending on whether the neurons i and j are excitatory (E) or inhibitory (I). In our experiments we used spiking neurons according to the standard leaky-integrate-and-fire (LIF) neuron model that are connected via dynamic synapses. The time course for a post-synaptic current is approximated by the equation

$$v(t) = w \cdot e^{-\frac{t}{\tau_{syn}}} \tag{3.2}$$

where w is a synaptic weight and τ_{syn} is the synaptic time constant. In case of dynamic synapses the "weight" w depends on the history of the spikes it has seen so far according to the model from [MWT98]. For synapses transmitting analog values (such as the output neurons in our experimental setup), synapses are simply modeled as static synapses with a strength defined by a constant weight w. Additionally, synapses for analog values can have delay lines, modeling the time a potential would need to propagate along an axon.

3.3 Experimental Setup

In this section we introduce the general setup that was used during our experiments to solve prediction tasks with real-world data from a robot. As depicted in Figure 3.2, such a network consists of three different neuron pools: (a) an input layer that is used to feed sensor data from the robot into the network, (b) a pool of neurons forming the LSM according to Section 3.2 and (c) the output layer consisting of readout neurons which perform a linear combination of the membrane potentials obtained from the liquid neurons.

For simulation within the training and evaluation the neural circuit simulator *CSim*[1] was used. Parameterization of the LSM is described below. Names for neuron and synapse types all originate from terms used in the *CSim* environment. Letters I and E denote values for inhibitory and excitatory neurons respectively.

To feed activation sequences into the liquid pool, we use *external input neurons* that conduct an injection current I_{inject} via *static analog synapses* (parameters are shown in Table 3.1) into the first layer of the liquid pool. Inspired from information processing in living organisms, we set up a cognitive mapping from input layer to liquid pool. The value of I_{inject} depends on the value of the input data, in this case the activation of each single visual sensor.

The liquid pool consists of *leaky integrate and fire neurons* – whose parameters are listed in Table 3.2 — grouped in an cuboid eight times six times three large, that are randomly connected via *Dynamic Spiking Synapses* (parameters are listed in Table 3.3), as described above.

[1]The software simulator *CSim* and the appropriate documentation for the liquid state machine can be found on the web page http://www.lsm.tugraz.at.

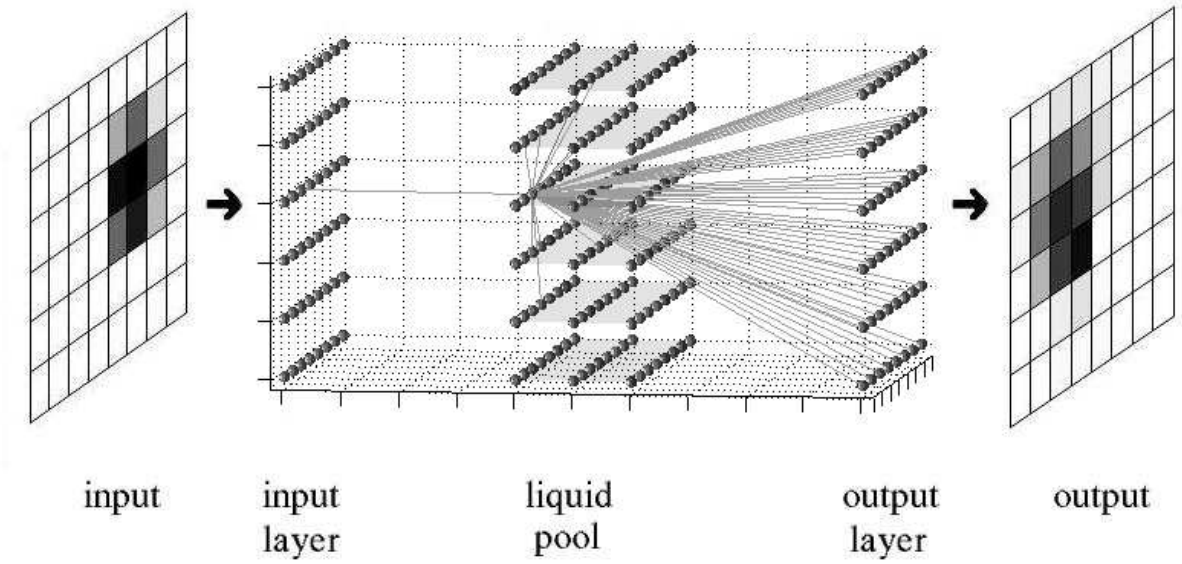

Figure 3.2: Architecture of our experimental setup depicting the three different pools of neurons and a sample input pattern with the data path overview. Example connections of a single liquid neuron are shown: input is received from the input sensor field on the left hand side and some random connection within the liquid. The output of every liquid neuron is projected onto every output neuron (located on the most right hand side). The 8 times 6 times 3 neurons in the middle form the ”liquid”.

The probability of a connection between every two neurons is modeled by the probability distribution depending on a parameter λ described in the previous section. Various combinations of λ (connection probability) and mean connection weights Ω (connection strength) were used for simulation. 20% of the liquid neurons were randomly chosen to produce inhibitory potentials. Figure 3.2 shows an example for connections within the LSM.

The information provided by the spiking neurons in the liquid pool is processed (read out) by *external output neurons* (V_{init}, $V_{resting}$, I_{noise} are the same as for the liquid neurons), each of them connected to all neurons in the liquid pool via *Static Spiking Synapses* (τ_{syn} = 3ms (EE) or 6ms (EI), $w = -6.73 * 10^{-5}$ (e.g., set after training), $delay_{mean}$ = 1.5ms (EE) or 0.8ms (EI) with $CV = 0.1$). The output neurons perform a simple linear combination of inputs that are provided by the liquid pool.

We evaluate the prediction approach by carrying out several experiments with real-world data in the RoboCup Middle-Size robotic soccer scenario. The experiments were conducted using a robot of the “Mostly Harmless” RoboCup Middle-Size team [FSW04b]. The task within the

I_{noise}	w_{mean}		$delay_{mean}$	
$[nA]$	-		$[ms]$	
	EE	EI	EE	EI
0	$3 \cdot 10^{-8}$	$6 \cdot 10^{-8}$	1.5	0.8

Table 3.1: Parameters for the static analog synapses which are used to feed input data into the LSM. 'EE' or 'EI' denotes whether the source and target neurons of a connection release excitatory or inhibitory action potentials, respectively. Covariance for $delay_{mean}$ is 0.1.

C_m	R_m	V_{thresh}	$V_{resting}$	V_{reset}	V_{init}	$T_{refract}$		I_{noise}	I_{inject}
$[nF]$	$[M\Omega]$	$[mV]$	$[mV]$	$[mV]$	$[mV]$	$[ms]$		$[nA]$	$[nA]$
						E	I		
30	1	15	0	$(13.8, 14.5)$	$(13.5, 14.9)$	3	2	0	$(13.5, 14.5)$

Table 3.2: Parameters for the leaky integrate and fire neurons comprising the liquid pool. Letters 'E' and 'I' indicate whether the neurons emit excitatory or inhibitory action potentials. (a, b) denotes an uniform distribution on the interval $[a, b]$.

experiments is to predict the movement of the ball in the field of view a few frames into the future. The experimental setup can be described as follows: The robot is located on the field and points its camera across the field. The camera is a color camera with a resolution of 320 times 240 pixel. The ball is detected within an image by simple color-blob-detection leading to a binary image of the ball. We can use this simple image preprocessing since all objects on the RoboCup-field are color-coded and the ball is the only red one. The segmented image is presented to the 8 times 6 sensor field of the LSM. The activation of each sensor is equivalent to the percentage of how much of the sensory area is covered by the ball.

We collected a large set of 674 video sequences of the ball rolling with different velocities and directions across the field. The video sequences had different lengths and contain images in 50ms time steps. These video sequences were transfered into the equivalent sequences of activation patterns of the input sensors. Figure 3.3 shows such a sequence. The activation sequences were randomly divided into a training set (85%) and a validation set (15%) used to train and evaluate the prediction. Training and evaluation was conducted for the prediction of 2 time steps (100ms), 4 time steps (200ms) and 6 time steps (300ms) ahead. The corresponding target activation sequences were simply obtained by shifting the input activation sequences 2, 4 or 6 steps forward in time.

Simulation for the training set was carried out sequence-by-sequence: for each collected ac-

	U_{mean}	D_{mean}	F_{mean}	$delay_{mean}$	τ_{syn}	C
con.	-	-	$[s]$	$[ms]$	$[ms]$	-
EE	0.5	1.1	0.05	1.5	3	0.3
EI	0.05	0.125	1.2	0.8	3	0.4
IE	0.25	0.7	0.02	0.8	6	0.2
II	0.32	0.144	0.06	0.8	6	0.1

Table 3.3: Parameters for the dynamic spiking synapses connecting the neurons within the liquid pool. 'EE', 'EI', 'IE' and 'II' denote whether the source and target neurons of a connection emit excitatory or inhibitory action potentials. Covariance for $delay_{mean}$ is 0.1.

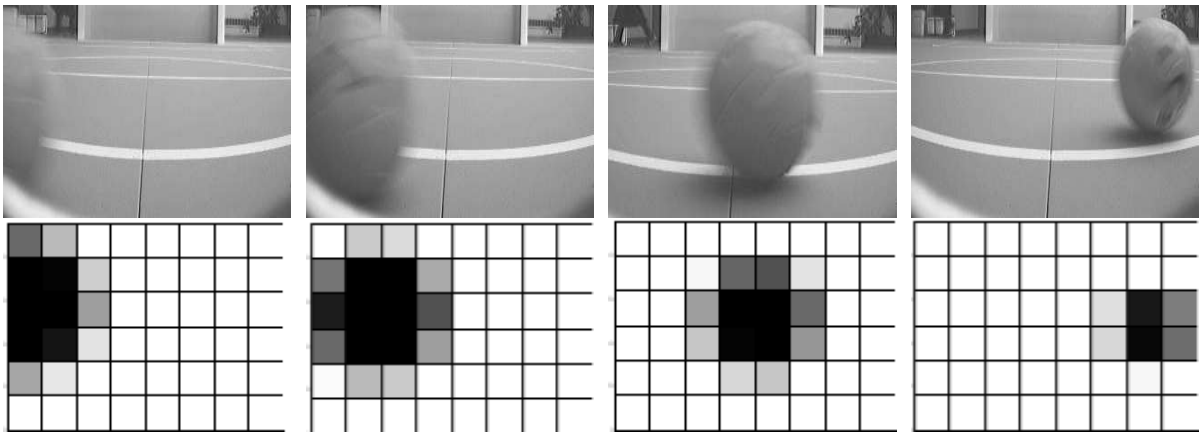

Figure 3.3: Upper Row: Ball movement recorded by the camera. Lower Row: Activation of the sensor field.

tivation sequence, the neural circuit is reset, input data were assigned to the input layer, recorders were set up to record the liquid's activity, simulation was started, and the corresponding recorded liquid activity are stored for the training part. The training was performed by calculating the weights[2] of all static synapses connecting each liquid neuron with all output layer neurons using linear regression.

Analogous to the simulation with the training set, simulation was then carried out on the validation set of activation sequences. The resulting output neuron activation sequences (*output sequences*) were stored for evaluating the network's performance.

[2]In fact also the injection currents I_{inject} for each output layer neuron was calculated. For simplification this bias was treated as the 0^{th} weight

3.4 Results

We introduce the mean absolute error and the correlation coefficient to evaluate the performance of the network. The mean absolute error is the positive difference between the activation values of target and output sequences of the validation set divided by the number of neurons in the input/output layer and the length of the sequence. This average error per output neuron and per image yields a reasonable measure for the performance on validation sets with different length. Figure 3.4 shows an example for a prediction and its error.

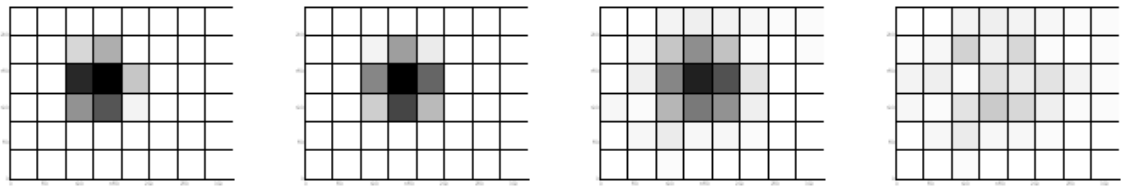

Figure 3.4: Sensor activation for a prediction one time step ahead. Input activation, target activation, predicted activation and error (left to right).

A problem which arises if only the mean absolute error is used for evaluation is that also networks with nearly no output activation produce a low mean absolute error — because most of the neurons in the target activation pattern are not covered by the ball and therefore they are not activated leading to a low average error per image. The correlation coefficient measures the *linear* dependency of two variables. If the value is zero two variables are not correlated. The correlation coefficient is calculated in similar way as the mean absolute error. Therefore the higher the coefficient the higher the probability of getting a correlation as large as the observed value without coincidence involved. In our case a relation between mean absolute error and correlation coefficient exists. A high correlation coefficient indicates a low mean absolute error.

In Figure 3.5 the mean absolute errors averaged over all single images in the movies in the validation set and the correlation coefficients for the prediction one time step (50ms) ahead are shown for various parameter combinations. The parameter values range for both landscapes from 0.1 to 5.7 for Ω and from 0.5 to 5.7 for λ. If both Ω and λ are high, there is too much activation in the liquid. Remember, λ controls the probability of a connection and Ω controls the strength of a connection. We assume that this high activity hampers the network making a difference between the input and the noise. Both values indicate a good area if at least one of the parameters is low. Best results are achieved if both parameters are low (e.g. Ω=0.5, λ=1.0). The figure clearly shows the close relation between the mean absolute error and the correlation coefficient. Furthermore, it shows the very good results for the prediction as the correlation coefficient is close to 1.0 for good parameter combinations.

It should be noted that the mean absolute error has to be used with caution. During calculation the value is divided by the number of neurons in the layer, so a lot of neurons that are not involved(the area is never reached by the ball, so it is never activated) contribute to the dividend. Especially when comparing two results which differ in prediction time, we neglect the mean absolute error from now on and focus on the correlation coefficient.

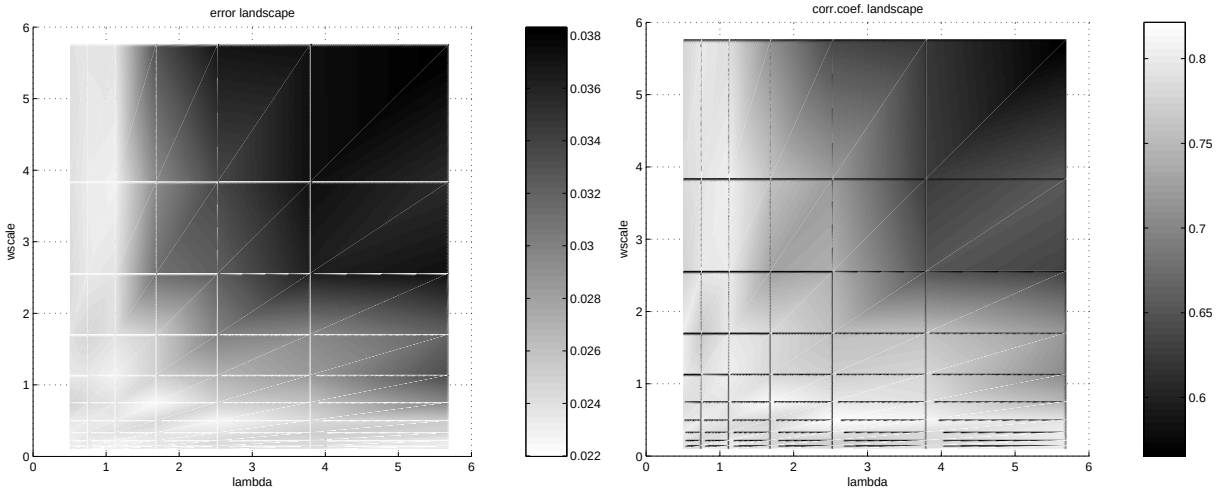

Figure 3.5: Mean absolute error landscape on the left and correlation coefficient on the right for a prediction one time step ahead. $\Omega(wscale)$ [0.1,5.7], λ [0.5,5.7]

We also compare the results achieved with two (100ms) and four (200ms) time steps predicted. In order to compare the results of both predictions for different parameter combinations, we use again a landscape plot of the correlation coefficients. Figure 3.6 shows the correlation coefficient for parameter values range from 0.1 to 5.7 for Ω and from 0.5 to 5.7 for λ. The regions of good results remain the same as in the one time step prediction. If at least one parameter — Ω or λ — is low the correlation coefficient reaches its maximum (about 0.7 at two time steps and about 0.5 at four time steps). With increasing Ω and λ, the correlation coefficients decrease again. We believe that the too high activation is again the reason for this fact. Not surprisingly the maximum correlation compared to the one step prediction is lower because prediction gets harder if the prediction time increases. Nevertheless, the results are good enough for reasonable predictions.

Figure 3.7 shows an example for the activations and the error for the prediction of two time steps ahead. It clearly shows that the center of the output activation is in the region of high activation in the input and the prediction is reasonable good. The comparison to Figure 3.4 also shows that the activation is more and more blurred around its center if the prediction time increases.

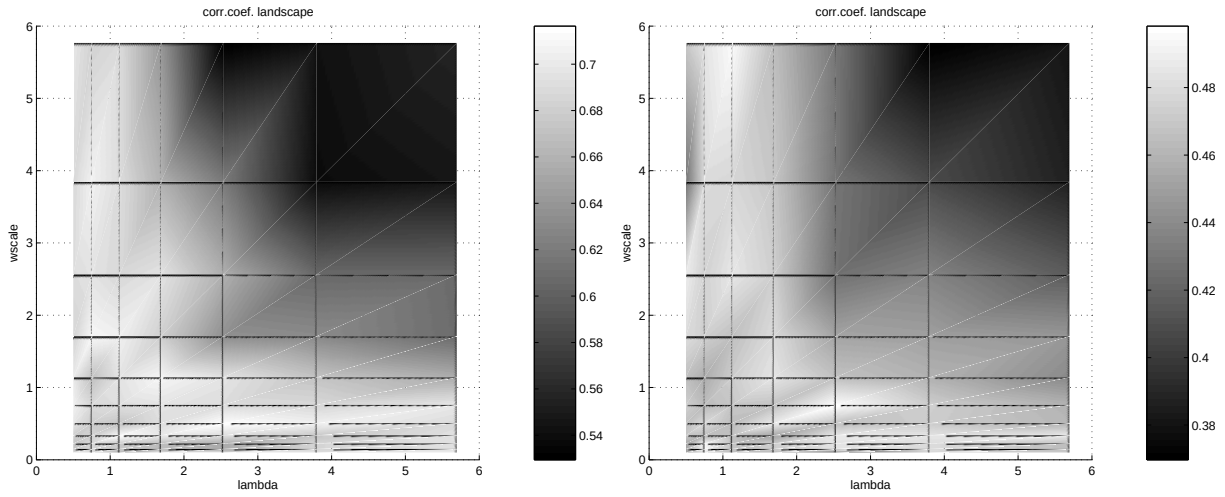

Figure 3.6: Correlation coefficient landscape for two time steps (100ms) on the left hand side and four time steps (200ms) on the right hand side.

Furthermore we confronted the liquid with the task to predict 300ms (6 time steps) without getting a proper result. We were not able to visually identify the ball position anymore. We guess this is mainly caused by the blur of the activation.

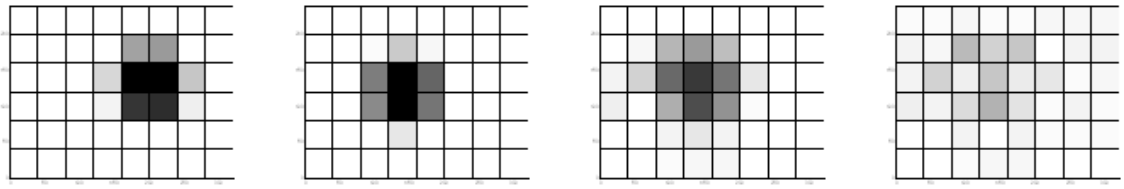

Figure 3.7: Sensor activation for a prediction two time steps ahead. Input activation, target activation, predicted activation and error (left to right). Parameter: Ω=1.0, λ=2.0

3.5 Discussion

In this chapter we described a biologically more realistic approach for the computation of time series of real world images. The LSM, a new biologically inspired computation paradigm, is used to learn ball prediction within the RoboCup robotic soccer domain. The advantages of the LSM are that it projects the input data in a high-dimensional space and therefore simple learning methods, e.g., linear regression, can be used to train the readout. Furthermore, the *liquid*, a

pool of inter-connected neurons, serves as a memory which holds the current and some past inputs up to a certain point in time (fading memory). Finally, this kind of computation is also biologically more plausible than other approaches like Artificial Neural Networks or Kalman Filters. Preliminary experiments within the RoboCup domain show that the LSM approach is able to reliably predict ball movement up to 200ms ahead. But there are still open questions. One question is how the computation is influenced by the size and topology of the LSM. Moreover, deeper investigation should be done for more complex non-linear movements, like balls bouncing back from an obstacle and for different ball velocities and viewing angles. Furthermore, it might be interesting to directly control actuators with the output of the LSM. We currently work on a goalkeeper, which intercepts the ball, controlled directly by the LSM approach.

Chapter 4

Intelligent Qualitative Control

In Chapter 2 it has been motivated that a combination of a reactive and a deliberative layer is the most appropriate concept for the control of an autonomous mobile robot in a dynamic environment. The existence of exogenous events makes dynamic environments unpredictable. Therefore, also the deliberative component has to have the capability to deal with dynamic unpredictable domains. Several such domains are used as common test-beds for the application of qualitative techniques to robots acting in dynamic environments, e.g. robotic soccer, tour guide robots or service and delivery robots. These domains come close to the real world where the gathered data are error prone, agents are truly autonomous, action execution regularly fails, and exogenous events are ubiquitous.

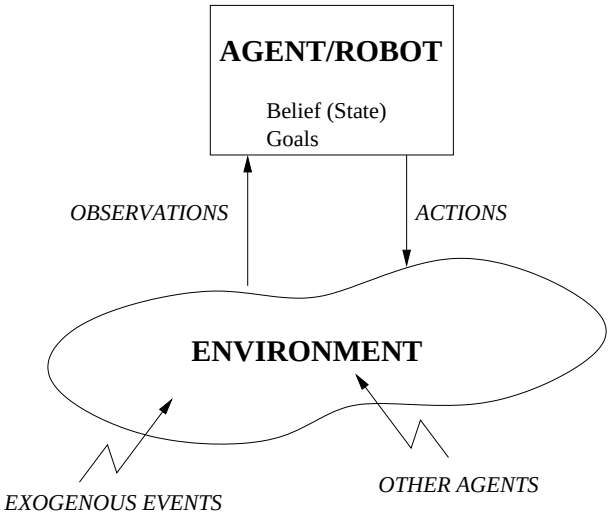

Figure 4.1: Interaction between agents/robots and their environment

Agents deployed in such domains have to interact with their environment. Figure 4.1 depicts

the interaction between the agents/robots and their environment. An agent has a belief about its environment and goals it has to achieve. Such beliefs are derived from domain knowledge and environment observations. While pursuing its goal by executing actions that influence the environment, the agent assumes these actions cause exactly the desired changes and that its belief reflects the true state of the environment. However, due to ambiguous or noisy observations and occlusions the belief of the agent and the state of the environment are not necessarily consistent. Furthermore, other agents or exogenous events may also affect the environment in an unpredictable way. Finally, actions might fail to achieve their desired effect. Clearly, an agent has to be able to cope with such influences in order to be able to successfully achieve a given goal. In this chapter a solution is presented which enables an agent to quickly react to such influences in order to be able to successfully achieve a given goal. The ideas of this chapter were also published in [FSW05].

To investigate the advantages of the proposed solution, experiments were conducted using the prior introduced robot architecture. On the software side a three-layered architecture is used that separates hardware interfaces, numerical and symbolic data processing. The symbolic layer hosts an abstract knowledge-base (belief), a planning system which is based on classical AI planning theories, and a plan executor. The representation language used is based on the well known STRIPS [FN71] representation language and incorporates numerous extensions thereof that have been presented in recent years, allowing the usage of first-order logic with only minor restrictions. A complete introduction into the topic of planning can be found in [GNT04].

The execution of a plan's actions is twofold. For one, on an abstract layer execution is supervised in a purely symbolic manner by monitoring conditions. On a numerical layer, where none of the abstract layer's symbols are known, a set of elementary behaviors corresponding to the abstract actions are executed. This behavioral approach for low-level action execution ensures that reactivity is achieved where needed, and incorporates tasks such as path planning or obstacle avoidance that are not of concern to the symbolic representation.

In this chapter, the idea of plan invariants as a means to supervise plan execution is presented. Plan invariants are conditions that have to hold during the whole plan execution. Consider a delivery robot, based on the above described architecture. Its task is to transport a letter from room A to room D. This task is depicted in Figure 4.2 and 4.3. The robot believes that it is located in room C, the letter is in room A and all doors are open. Its goal is that the letter is in room D. The robot might come up with the following plan fulfilling the goal: (1) move to $Room\_A$, (2) pick up letter L, (3) move to $Room\_D$ and (4) release letter L. In situation (a) no exogenous events occur, the belief of the agent is always consistent with the environment. Therefore, the robot is able to execute the plan and achieves the desired goal. In situation (b) the robot starts to execute the plan with action (1). Unfortunately, somebody closes the door to room D (2). As the robot is not able to open doors, its plan will fail. Without plan invariants the

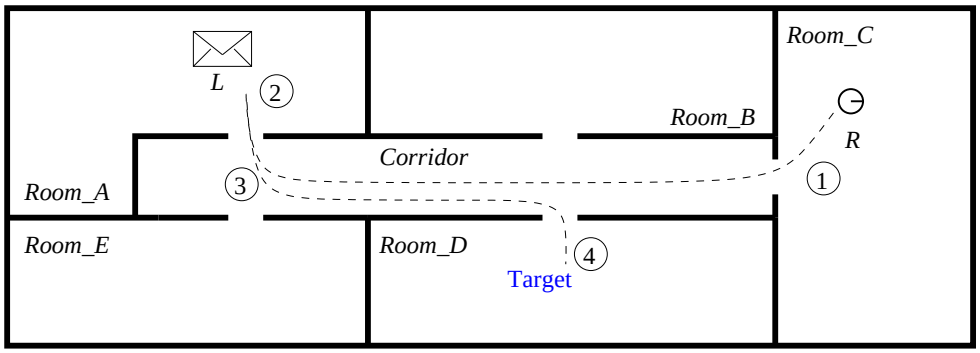

Figure 4.2: Successful execution of the plan: (1) move to $Room\_A$, (2) pick up letter L, (3) move to $Room\_D$ and (4) release letter L.

robot will continue to execute the plan until it tries to execute action (3) and detects the infeasible plan. If a plan invariant is used, e.g., room D has to be accessible, the robot detects the violation as it passes the closed door. Therefore, the robot is able to early detect invalid plans and to quickly react to exogenous events. Figure 4.4 depicts the temporal course of the two different plan executions.

In the next section the advantages of plan invariants in are discussed in more detail.

4.1 Plan Invariants

Invariants are facts that hold in the initial and all subsequent states. Their truth value is not changed by executing actions.

There is a clear distinction between these plan invariants to action preconditions, plan preconditions and invariants applied to the plan creation process. Action preconditions have to be true in order to start execution of an action. They are only checked once at the beginning of an action. Similarly, plan preconditions (i.e., initial state) are only checked at the beginning of plan execution. Thus, preconditions reflect conditions for points in time whereas invariants monitor time periods. In the past, invariants have been used to increase the speed of planning algorithms by reducing the number of reachable states, for an overview see [RH01]. An invariant as previously described characterizes the set of reachable states of the planning problem. A state that violates the invariant cannot possibly be reached from the initial state. For example, this has been efficiently applied to Graphplan [BF95] as described in [FL98, FL00]. Such invariants can be automatically synthesized as has been shown in [Rin00, KC92]. However, plan invariants are not only useful at plan creation time but also especially at plan execution time. To the best knowledge plan invariants have never been used to control plan execution.

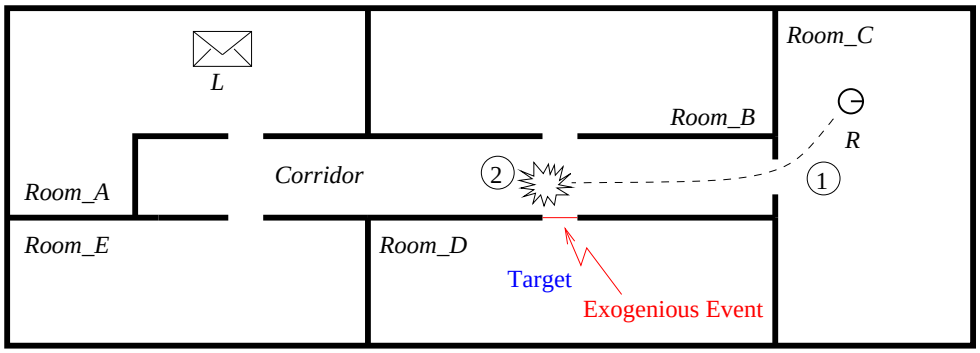

Figure 4.3: During execution of action (1) the exogenous event, close door to $Room\_D$, invalidates the plan. The robot stops the task because there is no possible plan as the target is not reachable anymore. In (2) the robot detects the closed door and the violation of the plan invariant ($accessible(Room\_D)$). Due to the application, of plan invariants the infeasibility of the plan is early detected.

There is a clear need for monitoring plan execution, because execution can fail for several reasons. Plan invariants can aid in early detection of non-executable actions, unreachable goals or infeasible actions. Reasons for failed plans are among others the following:

- **Actions are not executable**: The most obvious problem that can occur while executing a plan is that an action is not executable. The action's precondition tells when this has happened. If, however, an action that is a later part of the plan is not executable for reasons that are not influenced by other actions of the plan, precondition checking will not detect this until it is attempted to execute the failing action. An invariant, on the other hand, can be used to constantly verify this condition. Note that the condition to be checked must not be changed by any other action in the plan in order to use it with an invariant.

- **Goal unreachable**: Even if all not yet executed actions of the plan are executable, the goal state might still be unreachable. This could for example happen if the effects of previously executed actions are invalidated by some exogenous event. Furthermore, conditions not influenced by the actions that are part of the plan, e.g., conditions of the initial state, may change.

- **Goal not feasible**: As the environment is constantly changing, e.g., when considering a fast paced one such as robotic soccer, the aims a robot needs to pursuit can often change. In that case, all actions might be executable and the goal reachable and yet it might not be necessary that the agent achieves its task.

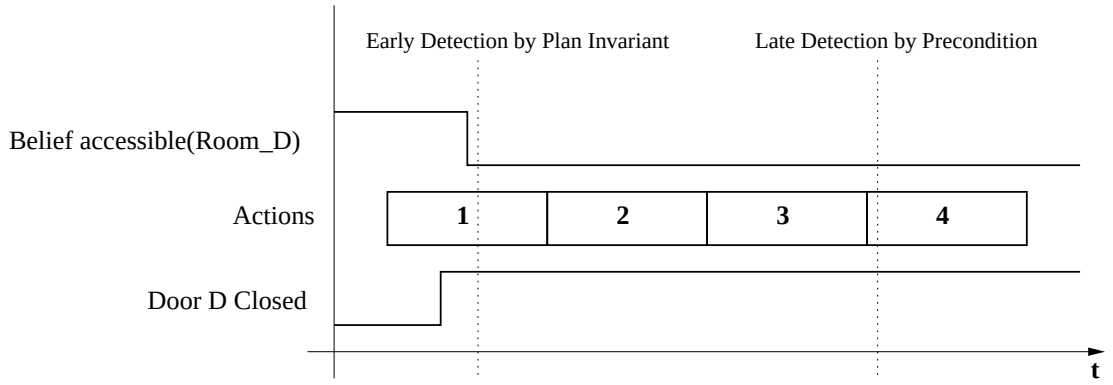

Figure 4.4: Plan execution for the deliver robot example in time and detection of invalid plans by checking plan invariants and by checking action's preconditions.

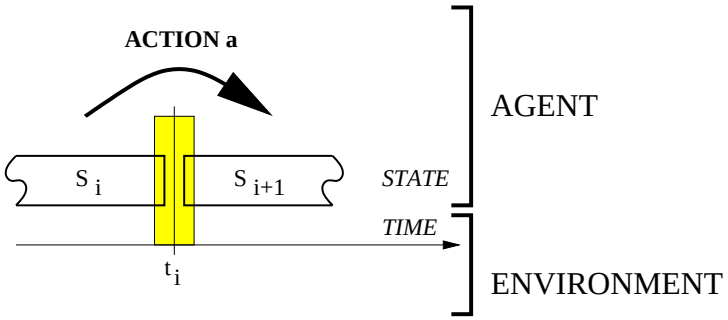

Figure 4.5: Action execution with respect to time for discrete actions.

Consider again the closed door example mentioned in the introduction. It is not practical to cover all such possible exogenous events, e.g., closed doors, within action preconditions, and neither is it within plan preconditions. On one hand this may be very exhaustive and, on the other hand this it constrains the general usability of actions.

4.2 Basic Definitions

Throughout this paper we use the following definitions which mainly originate from STRIPS planning [FN71]. A planning problem is a triple (I, G, A), where I is the initial state, G is the goal state, and A is a set of actions. A state itself is a set of ground literals, i.e., a variable-free predicate or its negation. The set of literals defines a conjunction of the literals. Each action $a \in A$ has an associated pre-condition $pre(a)$ and effect $\mathit{eff}(a)$ and is able to change a state via its execution. The pre-conditions and effects are assumed to be sets of ground literals. Execution

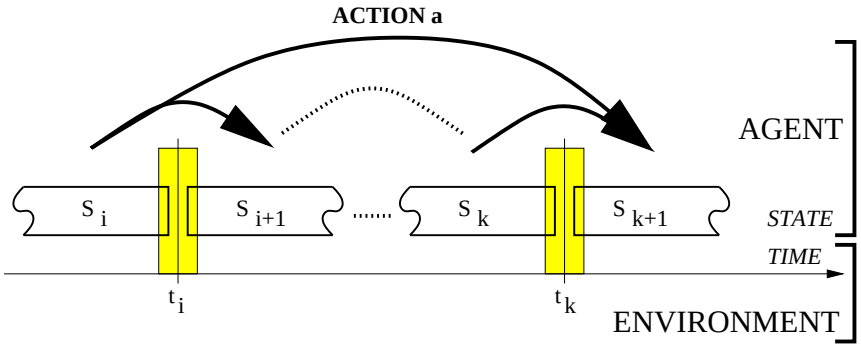

Figure 4.6: Action execution with respect to time for durative actions.

of an action a is started if its pre-conditions are fulfilled in the current state S. After the execution, all literals of the action's effect are elements of the next state S' together with the elements of S that are not influenced by action a. A plan p is a sequence of actions $[a_1, \ldots, a_n]$, that when executed starting with the initial state I results in goal state G.

For the delivery example the planning problem is defined as follows. The set of actions is A=$\{move, pickup, release\}$ with:

move(origin, dest):
pre: $accessible$(dest) $\wedge$ at(origin) $\wedge \neg$ at(dest)
eff: $\neg$ at(origin) $\wedge$ at(dest)

pickup(position,item):
pre: at(position) $\wedge$ $isat$(item,position) $\wedge \forall_{x:object(x)} \neg$ hold(x)
eff: $hold$(item)

release(position,item):
pre: at(position) $\wedge$ $hold$(item)
eff: $\neg$ $hold$(item) $\wedge$ $isat$(item,position)

The initial state is I:= $isat(L,\ Room\_A)\ \wedge\ at(Room\_C)$ and the goal is defined as G:= $isat(L,\ Room\_D)$. The names of constants and predicate are chosen quite intuitively. The predicate $accessible(p)$ indicates if a position p is accessible for the robot. The predicate $at(p)$ is evaluated true if the robot is actually at the position p. The predicate $ista(p,o)$ indicates if the object o is at the position p. If the robot actually holds the object o the predicate $hold(o)$ is evaluated true. The predicate $object(x)$ is true if the x is a moveable object. It has to be noted that all-quantifiers for finite domains can be easily transfered in a sentence with a conjunction of

the predicate instantiated with all members of the domain. Such a transfer is automatically done by the implementation of our planning system.

A plan can be automatically derived from a planning problem and there are various algorithms available for this purpose, refer to [Wel99] for an overview. For the delivery example a planner might come up with the plan p=[$move$($Room\_C$, $Room\_A$), $pickup$($Room\_A$,L), $move$($Room\_A$, $Room\_D$], $release$($Room\_D$,L)]. The planning problem makes some implicit assumptions for plan computation. First, it is assumed that all actions are atomic and cannot be interrupted. Second, the effect of an action is guaranteed to be established after its execution. Third, there are no external events that can change a state. Only actions performed by the agent alter states. Finally, it is assumed that the time granularity is discrete. Hence, time advances only at some points in time but not continuously. Such advances are triggered by the action execution.

In the simplest way plan execution is done by executing each action of the plan step by step without considering problems that may arise, e.g., a failing action or external events that cause changes to the environment. Formally, this simple plan execution semantics is given as follows (where $[\![\ ]\!]$ denotes the interpretation function):

$[\![< a_1, \ldots, a_n >]\!]\ S = [\![< a_2, \ldots, a_n >]\!]\ ([\![a_1]\!]\ S)$

$$[\![a]\!]\ S = \begin{cases} \mathit{eff}(a) \cup \{x | x \in S \wedge \neg x \notin \mathit{eff}(a)\} & \text{if } pre(a) \subseteq S \\ \textbf{fail} & \text{if } pre(a) \not\subseteq S \end{cases}$$

$[\![a]\!]$ **fail** = **fail**

Given the semantics definition of plan execution it can be stated what a feasible plan is.

Definition 1 *A plan* $p =< a_1, \ldots, a_n > | a_i \in A$ *is a feasible plan for a planning problem* (I, G, A) *iff* $[\![p]\!]\ I \neq$ **fail** *and* $[\![p]\!]\ I \supseteq G$.

Planning algorithms always return feasible plans. However, feasibility is only a necessary condition for a plan to be successfully executed in a real environment. Reasons for a plan to fail are:

1. An action cannot be executed.
   (a) An external event changes the state so that the pre-condition cannot be ensured.
   (b) The action itself fails because of an internal event, e.g., a broken part.
2. An external event changes the state of the world in a way so that the original goal cannot be reached anymore.
3. The action fails to establish the effect.

In order to formalize a plan execution in the real world, the following situation is assumed. A plan is executed by an agent/robot which has its view of the world. The agent can modify the state of the world via actions and perceives the state of the surrounding environment via sensors. The agent assumes that the sensor input is reliable, i.e., the perceived information reflects the real state of the world. This assumption is obviously not true, but is sufficient good approximation for most of the domains. Hence, during plan execution the effects of the executed actions can be checked via the sensor inputs. For this purpose, a global function *obs(t)* is assumed which maps a point in time t to the observed state. Note that the closed world assumption is used. Any predicate remains false until it is observed as true.

$$S \oplus obs(t) = obs(t) \cup \{l | l \in S \wedge \neg l \notin obs(t)\} \tag{4.1}$$

defines an update function for the agent's belief. The function returns all information about the current state that is available, i.e., the observations together with derived conditions during plan execution which are not contradicting the given observations.

In order to define the execution of an action in the real world, two cases need to be distinguished. Actions can last a fixed, known time. In this case, execution is considered done after that time has elapsed. For a RoboCup agent, such an action could be the act of kicking the ball. On the other hand, actions can continue indefinitely, e.g., a move action in a dynamic environment can take unexpectedly long if changes in the dynamic environment require detours. Execution of such an action is considered to be finished as soon as its effect is fulfilled. Following the terminology previously used in [Nil94], actions with fixed duration are called *discrete*, and indefinitely continued actions are called *durative*.

Figure 4.5 and Figure 4.6 depict the action execution with respect to time. A discrete action a is executable if its precondition $pre(a)$ is satisfied in state S_i, where a state $S_i = S_{i-1} \oplus obs(t_{i-1})$.

An action lasts for a given time and tries to establish its effect $\mathit{eff}(a)$ in the succeeding state S_{i+1}. A durative action a is also executable if its precondition $pre(a)$ is satisfied in state S_i. In contrast to discrete actions, a durative action a is executed until its effect $\mathit{eff}(a)$ is established in some following state S_{k+1}. At each time step t_j, $i \leq j < k+1$ a new observation is available, a new state S_{j+1} is derived $S_{j+1} = S_j \oplus obs(t_j)$. For each state S_{j+1} the condition $\mathit{eff}(a) \subseteq S_{j+1}$ is evaluated. A durative action can possibly last forever if it is impossible to establish the effect $\mathit{eff}(a)$.

For discrete actions, execution semantics can be written as follows:

$$\text{if } \textbf{discrete}(\text{a}) \text{ then } [\![a]\!]\,(S) = \\ = \begin{cases} S \oplus obs(t) & \text{if } \mathit{eff}(a) \subseteq (S \oplus obs(t)) \\ [\![\mathbf{exec}]\!](a, S \oplus obs(t)) & \text{if } \mathit{pre}(a) \subseteq (S \oplus obs(t)) \wedge \mathit{eff}(a) \not\subseteq (S \oplus obs(t)) \\ \textbf{fail} & \text{otherwise} \end{cases} \tag{4.2}$$

In the above definition of the plan execution semantics for discrete actions, we can distinguish three cases. The first line of the definition handles the case where the effect is fulfilled without the requirement of executing the action a. In the second line, the action a is executed which is represented by the $\mathbf{exec}(a, S)$ function. t is the time after executing the action.

$$[\![\mathbf{exec}]\!]\,(a, S) = \begin{cases} \mathit{eff}(a) \cup \left\{ x \;\middle|\; \begin{array}{c} x \in S \wedge \\ \neg x \notin \mathit{eff}(a) \end{array} \right\} & \text{if action } a \text{ is executed} \\ \textbf{fail} & \text{otherwise} \end{cases} \tag{4.3}$$

$\mathbf{exec}(a, S)$ returns **fail** if the action a was not executable by the agent/robot in state S. If action a is executed $\mathbf{exec}$ returns the effect of the action $\mathit{eff}(a)$ unified with all literals of state S not negated by $\mathit{eff}(a)$.

The last line of the execution semantics states that it returns **fail** if the precondition of the action is not fulfilled. The action *release* is an example for a discrete action. Once the action is triggered, it either takes a certain amount of time to complete or it fails.

For durative actions, execution semantics can be written as follows:

$$\text{if } \textbf{durative}(\text{a}) \text{ then } [\![a]\!]\,(S) = \begin{cases} S \oplus obs(t) & \text{if } \mathit{eff}(a) \subseteq (S \oplus obs(t)) \\ [\![a]\!]'\,(S \oplus obs(t)) & \text{if } \mathit{pre}(a) \subseteq (S \oplus obs(t)) \wedge \\ & \quad \mathit{eff}(a) \not\subseteq (S \oplus obs(t)) \\ \textbf{fail} & \text{otherwise} \end{cases} \tag{4.4}$$

with

$$[\![a]\!]'\,(S) = \begin{cases} S & \text{if } \mathit{eff}(a) \subseteq S \\ [\![a]\!]'\,(S \oplus obs(t')) & otherwise \end{cases} \tag{4.5}$$

The precondition of a durative action is checked only at the beginning of the action. It is assumed that one recursion of a durative action (Equation 4.5) lasts for a time span greater than zero. t' is the time step after the execution of one loop of the action. The action *move* is an example for a durative action, as it is executed until the robot reaches its destination. This may take different amounts of time or possibly may never occur.

Given a plan and a real-world environment it can now be defined what it means to be able to reach a goal after executing a plan.

Definition 2 *A plan $p = [a_1, \ldots, a_n]$ for a given planning problem (I, G, A) is successfully executed in a given environment if $[\![[a_1, \ldots, a_n]]\!](I) \supseteq G$.*

Obviously, there is a relationship between feasible plans and executed plans as stated in the following theorem.

Theorem 1 *A plan p for a planning problem (I, G, A) is successfully executed in a given environment with observations obs if (1) the plan is feasible, and (2) for every execution of an action $a \in p$ the condition eff $(a) \subseteq [\![a]\!](S)$ is satisfied.*

Theorem 1 states a condition for the successful execution of a plan. If every execution of an action has the desired effect which can be observed and the plan is feasible, then the goal must be reached.

4.3 Extended Planning Problem

As outlined in Section 4.1, plan invariants are a useful extension to the planning problem. The addition of an invariant to a planning problem results in the following definition:

Definition 3 *An extended planning problem is a tuple (I, G, A, inv) where inv is a logical sentence which states the plan invariant.*

A plan p for an extended planning problem is created using any common planning algorithm. We call the pair (p, inv) extended plan.

The plan invariant has to be fulfilled until the execution of the plan is finished (either by returning the goal state or **fail**). A plan invariant is a more general condition for feasible plans. It allows for considering exogenous events and problems that may occur during execution, e.g., failed actions. Automatic generation of such invariants is questionable. For a deeper discussion refer to Section 4.4. Invariants may represent knowledge that is not implicitly contained in the planning problem, and thus cannot be automatically extracted from preconditions and effect descriptions. An open question is how more knowledge about the environment (e.g., modeling physical laws or the behavior of other agents) and an improved knowledge representation would enable automatic generation of plan invariants.

The execution semantics of such an extended plan can be stated using $\|$ to denote parallel execution:

$$[\![(p, inv)]\!]\ (S) = [\![p]\!]\ (S) || [\![inv]\!]\ (S) \tag{4.6}$$

Communication between statements executed in parallel is performed through obs, S and the state of plan execution.

The semantics of checking the invariant over time is defined as follows:

$$[\![inv]\!](S) = \begin{cases} [\![inv]\!](S \oplus obst(t)) & \text{if } inv \cup S \not\models \bot \\ \textbf{fail} & otherwise \end{cases} \tag{4.7}$$

where S is the current belief state of the agent and $obs(t)$ results in a set of observations at a specific point in time t. Hence, the invariant is always checked unless it contradicts the state of the world obs or the agent's belief S. For the delivery example inv = $accessible$($Room\_D$) $\wedge$ ($accessible$($Room\_A$) $\vee$ $hold$($letter$)) would be a feasible invariant. The invariant states that as long as the robot does not hold the letter, $Room\_A$ has to be accessible. Room_D has to be accessible during the whole plan execution.

Definition 4 *An extended plan $p = ([a_1, \ldots, a_n], inv)$ is a feasible extended plan for a planning problem (I, G, A) iff $[\![p]\!]\ I \neq$ **fail** and $[\![p]\!]\ I \supseteq G$, and all states that are passed by the plan the invariant must hold, i.e., $\forall_{i=0}^{n}([\![a_1, \ldots, a_i]\!](I) \cup inv) \not\models \bot$.*

Feasibility is again a necessary condition for extended plans to be executable. But it is not guaranteed that the agent will reach its goal. Hence, it must be guaranteed that the invariant does not contradict any state that is reached during plan execution. We now can easily extend Definition 2 for extended plans.

Definition 5 *An extended plan $p = ([a_1, \ldots, a_n], inv)$ for a given planning problem (I, G, A) is successfully executed in a given environment if $[\![(< a_1, \ldots, a_n >, inv)]\!](I) \supseteq G$.*

Theorem 2 *An extended plan $p = ([a_1, \ldots, a_n], inv)$ for a planning problem (I, G, A) is successfully executed in a given environment with observations obs if (1) the plan is feasible, (2) $\forall_{i=0}^{n}([\![a_1, \ldots, a_i]\!](I) \cup inv) \not\models \bot$ and (3) the set of believed facts resulting from execution of plan p with simple plan execution semantics is a subset of the set of believed facts resulting from execution in a real-world environment.*

Regarding Theorem 2 (3), in real-world environments, observations lead to believed facts that are not predictable from the plan execution, hence $[\![a]\!](S)$ differs.

Corollary 3 *Every feasible extended plan for a planning problem* (I, G, A) *is a feasible plan for the same planning problem.*

The prove is quite intuitive because the invariant does not affect the planning process.

Concluding the execution of a plan does not relieve an agent of its duties. If the plan execution succeeds, a new objective can be considered. If plan execution fails, alternative designations need to be aimed at. Not all possible goals might be desirable, therefore a condition that decides about execution is needed. This condition needs to be valid from the beginning of plan creation to the initiation of plan execution, hence the initial state I needs to fulfill this condition, the *plan problem precondition.* An agent is given a set of alternative planning problems $P_1, \ldots, P_n$ and nondeterministically picks one out of these that has a satisfied precondition C_i thus deriving an extended planning problem (I, G_i, A, inv).

$$\Pi = \left\{ \begin{array}{l} C_1 \rightarrow (I, G_1, A, inv) \\ ... \\ C_n \rightarrow (I, G_n, A, inv) \end{array} \right\}. \tag{4.8}$$

The knowledge base of an agent Π comprises of all desired reactions of the agent to a given situation. The preconditions trigger sets of objectives the agent may pursue in the given situation.

The execution semantics of this set of planning problems can be stated as follows:

$[\![\Pi]\!]\ (I) =$
do for ever
select (I, G_i, A, inv) **when** $S \models C_i$
$p_i =$ **generate_plan**(I, G_i, A, inv)
$[\![(p_i, inv_i)]\!]\ (S)$
end do;

The function **generate_plan** generates a feasible plan. The plan could be generated by using any planning algorithm. The use of pre-coded plans is also conceivable. The function **select** nondeterministically selects one planning problem of the set of planning problems whose precondition is fulfilled. A heuristic implementation of the function is conceivable, if some measure of the performance/quality of the different planning problems is available.

The semantics of the extended planning problem and the plan execution define a semantics of a general symbolic program language, if some minor restrictions are applied. Plans are similar to sequential programs in the imperative programming. The actions pose as statements. If pre-coded plans (equivalent to subroutines), pre-coded invariants and a defined heuristic selection function are used, the behavior of agents acting in a dynamic environment can be described in a

general symbolic way. Due to the possibility of the definition of plan invariants, the programmer is able to ensure that the agent is reactive to exogenous or unpredicted events, without coding each possible exception.

Plans are similar to sequential programs. If only pre-coded plans and a heuristic selection function are used, the extended planning problem and the execution semantics defines a general symbolic programming language for agents acting in dynamic environments.

The extended planning problem introduced above can be used for a definition of a pure symbolic easy to use programming language for agents. The programmer can easily define which tasks the agent should achieve (goal definition) and when it should pursuit the goal (plan preconditions). Furthermore, the programmer is able to define general conditions for the task execution (plan invariants). The programmer also may provides pre-calculated plans to the agent, which are equivalent to sequential subroutines.

4.4 Automated Generation of Plan Invariants

In the previous sections it has been shown that plan invariants are a appropriate mechanism to ensure and improve the plan execution in dynamic environments. Furthermore, it has been shown how these plan invariants can be used in execution monitoring and agent programming. But so far it was assumed that for each plan or planning problem the appropriate plan invariant is provided. In most of the cases, these invariants are hand-coded. But the hand-crafted generation of invariants is exhaustive and sometimes far from being trivial. Therefore, an automated generation of the invariants is desirable.

The problem of execution monitoring is not new. In the early 70's, Fikes already presented a monitoring mechanism for the STRIPS planning framework used in the control of the robot Shakey [FN71]. The idea was that a special condition called *kernel* is attached to each action in a plan. Kernels are necessary conditions that ensure that if the plan is further executed it may reach the goal state. Fikes initiated re-planning if the kernel of an action in a plan was not satisfied, because the goal state was not reachable anymore. The generation of the kernel is performed in the following way:

computeKernel(G, P)
  $n =$ **length**(P)
  $K_{n+1} = G$.
  For $i = n : 1 :$
    $K_i = (K_{i+1} \setminus \mathit{eff}\,(a_i)) \cup pre(a_i)$
  return K

The algorithm starts with the goal state and goes backwards through the action of the plan until it reaches the first action. The kernel K_{n+1} is set to goal state G, where n is the number of actions in the plan. The kernel K_i for an action a_i is calculated in the following way. First the algorithm removes those literals of the successor kernel K_{i+1} which are altered by the effect of the action a_i. Afterwards, all literals of the precondition of the a_i are added to this expression. This expression forms the kernel K_i and is attached to the action a_i. Table 4.1 depicts the calculated kernels for the plan of the delivery robot example.

K_1:	accessible(Room_D) $\wedge \neg$ at(Room_D) $\wedge \neg$ at(Room_A) $\wedge$ isat(letter, Room_A) $\wedge$ $\wedge \forall_{x:object(x)} \neg$ hold(x) $\wedge$ accessible(Room_A) $\wedge$ at(Room_C)
K_2:	accessible(Room_D) $\wedge \neg$ at(Room_D) $\wedge$ at(Room_A) $\wedge$ isat(letter, Room_A) $\wedge$ $\wedge \forall_{x:object(x)} \neg$ hold(x)
K_3:	hold(letter) $\wedge$ accessible(Room_D) $\wedge \neg$ at(Room_D) $\wedge$ at(Room_A)
K_4:	at(Room_D)) $\wedge$ hold(letter)
K_5:	isat(letter,Room_D)

Table 4.1: Calculated kernels for the deliver robot example.

During the execution of the plan, the truth-value of the kernel K_i is checked prior the execution of action a_i of the plan. If the kernel is satisfied, the execution of the plan is continued. Otherwise a re-planning is initiated. In [FHN81] the plan execution furthermore has the possibility to skip actions of the plan. If a kernel of a successor action in the plan is also satisfied by chance, the plan executor simple skips the current action and continues the plan execution with the later action. Furthermore, an efficient mechanism to monitor the whole set of kernels is provided in the work of Fikes.

The idea of kernels is somehow similar to the proposed plan invariants. The kernels provide a simple mechanism to monitor plan execution. Moreover, a simple algorithm for the automated generation of kernels for a plan is available. The main difference between plan invariants and kernels is that the kernels are different for each action in a plan. The proposed plan invariant remains the same for the whole plan. Furthermore, kernels represent only the precondition and effects of the actions of a plan. Plan invariants in contrast should encounter further the structure of the task/goal and the calculated plan.

Consider the following example of the presentation of a scientific paper at a conference. The usual way of presenting a paper starts with the submission of a paper to a conference. If the paper is once accepted, one registers for the conference. Therefore, the acceptance of the paper and the registration are preconditions for the task of presenting a paper. The obvious goal of the task is to give the talk at the conference. An appropriate plan might be to go first to the airport and fly to the city the conference takes place. Then take a taxi to the conference venue. Finally, if

the slides are prepared one is able to give the talk. But what happens if the conference session is canceled for some reason. It makes no sense to enter the plane, if this information is available in advance. But it also makes no sense to put all this information into the preconditions of actions like flying from A to B. This would constrain the general usability of an action. Therefore, for the automated generation of serious general plan invariants for different tasks a general knowledge and description of the structure of the task and reasoning capabilities are required.

4.5 Related Research

Invariants for planning problems have previously been investigated within the context of planning domain analysis. Planning domain descriptions implicitly contain structural features that can be used by planners while not being stated explicitly by the domain designer. These features can be used to speed up planning. For example, Kautz and Selman [KS98] used hand-coded invariants provided as part of the domain description used by Black-box, as did McCluskey and Porteous [MP97]. The use of such constraints has been demonstrated to have a significant impact on planning efficiency [GS98]. Such invariants can be automatically synthesized as has been shown in [Rin00, KC92, FL98]. Even temporal features of a planning domain can be extracted by combining domain analysis techniques and model checking in order to improve planning performance [FLBM01]. Also noteworthy is Discoplan [GS00], a system that uses domain description in PDDL [FL03] or UCPOP [PW92] syntax to extract various kinds of state constraints that can then be used to speed up planning. Any forward- or backward-chaining planning algorithm can be enhanced by applying such constraints, e.g. Graphplan [BF95], as described in [FL00]. However, in [BMM98] Baioletti, Marcugini and Milani suggest that such a constrained planning problem can be transformed to a non-constrained planning problem, which allows the application of any common planning algorithm.

In [Dij76] Dijkstra introduced the concept of guarded commands by using invariants for statements in program languages. This concept is similar to our proposed method except that we use it for plan execution.

The logic programming language Golog [LRL$^+$97] is based on Reiter's variant of the Situation Calculus. It is a second-order language which enables reasoning about actions and their effects. Golog and its derivate have been successfully used for the deliberative control of agents and robots. The advantage of Golog and its successors is that it combines logic interference, reasoning and planning with imperative control constructs like loops, conditionals and recursive procedures. Furthermore, it supports less standard constructs like nondeterministic action selection. Therefore, Golog forms a powerful and flexible logic-based programming language for agents and robots.

In [FFL04] DTGolog has been used to control robots in the RoboCup Middle-Size scenario.

DTGolog is an extension of Golog which uses decision-theoretic planning. Because of the properties of decision-theoretic planning like the reasoning about different outcomes of an action DTGolog is more suitable for dynamic environments like robot soccer. The authors extended the DTGolog framework with a mechanism for the monitoring action execution. Decision-theoretic planning uses models for actions which are able to predict how the state of the world evolves during the execution of the action. These models are similar to the effects in STRIPS planning. The predictions of the models are used during the planning process. Furthermore, the predictions are attached to the corresponding actions in the plan. These predictions are logical sentences and are called markers. The markers are monitored during the execution of an action. If the markers deviate from the actual state of the world the action is unable to fulfill its desired objective. The reason for this could be that the action failed or an external event have changed the world in an undesired way. In this case the execution of the action is interrupted. This is a simple mechanism for monitoring the execution of actions and plans. The idea is similar to the plan invariants but the marker mechanism ensures only the execution of a single action and not the execution of an entire plan.

In [BAB+01] Beetz and colleagues present their control architecture for the tour-guide robots Rhino and Minerva. The architecture comprises a plan-based high-level control called the structured reactive controller. Robustness and adaptability for unforeseen situation in the plan execution is achieved by the use of prediction models and the reactive plan language (RPL) which provides statements for robust execution.The structured reactive controller monitors the execution of plan by the the use of models of the behavior of the actions. In case of a problem or a failure the controller is able to modify plans at the fly in order to react to such situations. Furthermore, the controller is able to execute concurrent plans and if needed postpone of one plan the execution until a needed condition is satisfied again. Moreover, the architecture is able to learn action models in order to improve the planning and the plan execution. In [Bee02] Beetz gives a deeper discussion of the plan-based control of mobile robots.

4.6 Discussion

In this section a framework for executing plans in a dynamic environment has been presented. The framework was implemented for the autonomous robotic platform introduced in Chapter 2. The implementation was used in the RoboCup robotic soccer domain which led to promising results. Furthermore, the operational semantics of the framework has been discussed and it has shown under which circumstances the framework represents a language for representing the knowledge of an agent/robot that interacts with a dynamic environment but follows given goals. A major contribution of this section is the introduction of plan invariants which allow for representing knowledge that can hardly be formalized in the original STRIPS framework.

Summarizing, the main advantages gained by the use of plan invariants are:

- **Early recognition of plan failure**: The success of an agent in a real-world an environment is crucially influenced by its ability to quickly react to changes that influence its plans.
- **Long-term goals**: Plan invariants can be used to verify a plan when pursuing long term goals, as the plan's suitability is permanently monitored.
- **Conditions not influenced by the agent**: Plan invariants can be used to monitor conditions that are independent of the agent. Such conditions are not appropriate within action preconditions.
- **Exogenous events**: It is usually not feasible to model all exogenous actions that could occur, but plan invariants can be used to monitor significant changes that have an impact on the agent's plan.
- **Cooperation**: In order to successfully cooperate with other planning agents it is necessary to monitor the behavior of cooperating agents. For this task, plan invariants are perfectly suited.
- **Intuitive way to represent and code knowledge**: As the agent's knowledge commonly has to be defined manually it is helpful to think of plan preconditions (the situation that triggers the plan execution) and plan invariants (the condition that has to stay true at all times of plan execution) as two distinct matters.
- **Durative actions**: Plan invariants can be used to detect invalid or unsuitable plans during execution of durative actions. Durative actions, as opposed to discrete actions, can continue indefinitely. For example, a patrol robot in an office building can continually execute an action to wander around and to observe. As execution of this action does not finish, how does it detect a low battery and it would be more appropriate to drive to the charging station? Again, plan invariants offer a convenient solution.

Up to now the used plan invariants are hand-coded. It was motivated that plan invariants are able to encounter a much wider area of domain knowledge than action invariants and preconditions. For the desired automated generation of plan invariants further research is needed in the domain of descriptions of the structure of tasks and the reasoning about such descriptions.

Chapter 5

Bridging the Qualitative and the Quantitative World

Consider a robot that has to provide a certain task at a certain point in time. This robot has to have a knowledge about the physical world not only in terms of quantitative measurements like probability distributions of its location but also in terms of qualitative facts like predicates stating that a ball is in reach. This qualitative representation is necessary for computing actions in order to fulfill the task. Of course this picture of an autonomous agent assumes a symbolic reasoning engine on top which is used to handle high-level control in contrast to low-level control structures which can be implemented as reactive systems. Figure 5.1 shows the relation between the different representations of knowledge.

At the first sight the mapping of quantitative information to its qualitative representation seems not to be big deal and in some cases this is true. For example, when dealing with control systems for a plant with a limited (and known) number of possible interactions with the environment, the mapping problem can be solved by applying the right thresholds and filters. However, in applications like the robotic domain with unpredictable interactions between the robot and its environment the situation changes. Consider for example changes in the light condition. These changes have a substantial impact on the visual perception of the robot. Hence, an object which was within a one meter distance before changing the density of light maybe perceived at a higher distance afterwards although the real situation of the relationship between the object and the robot has not changed. Hence, the robot changes its internal state and may choose different actions. A more severe situation can happen when environmental changes cause the robot to switch between two contradicting states, e.g., object in reach and out of reach, which prevents the robot from taking meaningful actions. In this chapter we will discuss these problems in more detail and will present a approach which will decrease the impact of such effects. Parts of this chapter previously were published in [SWW05].

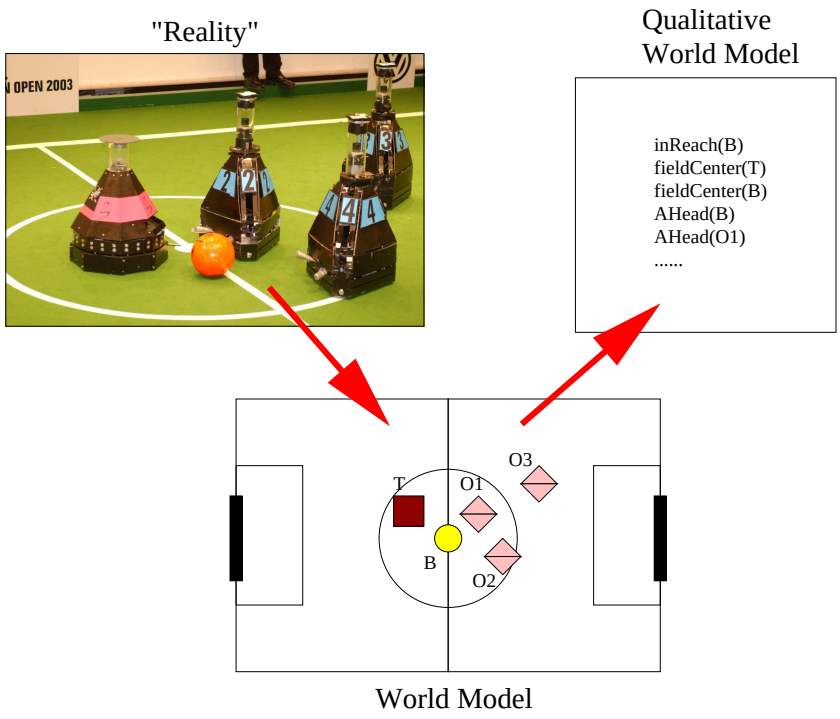

Figure 5.1: From the real world to its qualitative representation.

The qualitative mapping problem as described before is mainly caused by unreliable perception. Hence, one solution would be to improve the perception algorithms, e.g., the computer vision system, to make it less sensitive to changes of environmental parameters like light conditions. However, the mapping problem itself will not be solved. For example assume a perfect perception system and two robots playing soccer. The situation starts and our robot assumes the ball in reach but the opponent robot kicks the ball slightly. Hence, the situation changes and the ball is no longer in reach. Because of the underlying definition of in reach this might mean that the ball is now more than one meter apart from our robot which is also the case for a distance of let us say one meter and one centimeter. In both situations, we would expect our robot to take the same actions but because of a difference of one centimeter and the use of a sharp boundary the perceived world is different and thus the actions as well. A solution for this problem could be the introduction of new landmarks. In our example, a new predicate for almost reachable can be introduced. This kind of solution can be compared with solutions for the problem of finding the right qualitative reasoning model for a certain task. Sachenbacher and Struss [SS01] proposed such a solution.

The drawback of using only the quantitative values for making a decision in a specific situation at a given point in time becomes obvious when considering the soccer situations of Fig-

ure 5.2. All of the three soccer situations are different with respect to their corresponding quantitative model. However, from a high-level view situations (a) and (b) become equivalent because in both the ball B is in reach of player T and the way to the goal is not blocked by the opponent goalie O. Note that it seems that (a) and (b) are different because of the different distances to the goal but for the player's T view it makes (almost) no difference. Situation (c) is a different situation as the way to the goal is blocked.

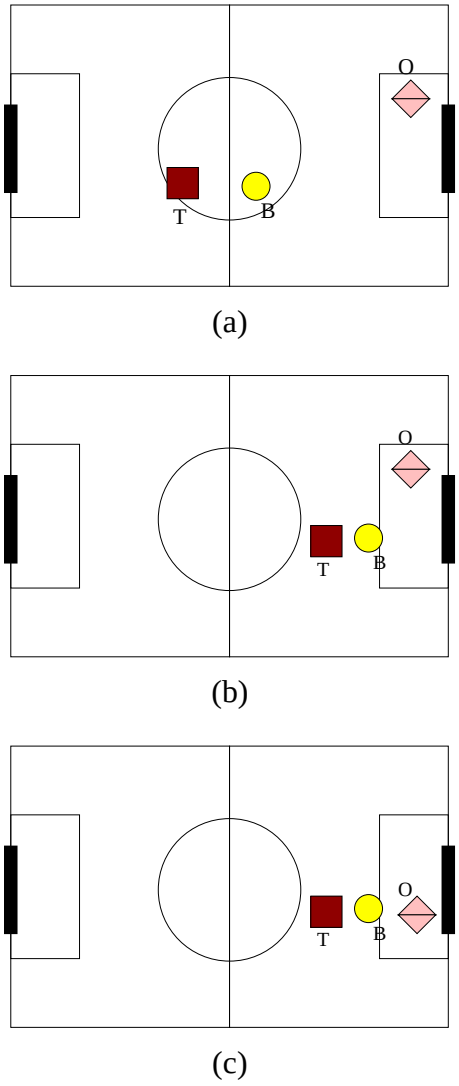

Figure 5.2: Three situations in robotic soccer. Quantitatively all three situations are different but qualitatively situation (a) and (b) are equivalent. Such relations are part of a qualitative model.

Although, the qualitative mapping problem can be theoretically solved by using perfect sensors and qualitative modeling techniques, there is still a need for a practical solution. Perfect

sensor input is not available and there is no indication that this problem will be solved soon. This holds especially for visioning systems. Hence, there is a requirement to overcome the problem. In this paper we follow the hysteresis approach from [FSW04a]. We introduce the problem again, present a practical solution in terms of predicate hysteresis, discuss experimental results and open issues. The experiments indicate that the use of hysteresis really improves the overall behavior.

5.1 Symbol Grounding and Action Selection

In order to create a continuous model of the real environment, the perceptions from different sensors (e.g., camera, odometry) are fused. The resulting model contains the positions of objects on the field: the ball, the two goals, and the players. There are different methods for creating world models in dynamic and nondeterministic environments; we use a Kalman filter [May90] for predictions of object positions and sensor fusion.

This purely quantitative model is transformed to an abstract world model, the *knowledge base*, which is expressed by means of a set K of ground predicates. The knowledge base, which is basically a conjunction of ground atoms, is the source for the qualitative reasoning which is performed by the *planner* [Fra03]. The planner is the strategy layer of the control software for our soccer robots, and its main responsibility is the selection of actions which shall be executed next. The Planner makes use of classic AI planning for creating plans at runtime. It is based on the STRIPS representation language [FN71].

This approach has, compared to reasoning based on continuous data, many advantages. Among others, a qualitative model has only a finite number of possible states, and qualitative models are able to cope with uncertain and incomplete knowledge. Another reason is the fact that the programming of the robot is simplified and can also be done by human operators who have no programming skills. The knowledge and the strategy can be neatly expressed in logical formulas.

As already explained, the knowledge of the robot is expressed using ground predicates. The interpretation of a n-ary predicate $p \in P$ relies on the continuous world model M. It can be formalized as follows:

$$I(p(O^n), M) = \begin{cases} true & \text{if } COND_p(O^n, M) = true \\ false & \text{otherwise} \end{cases} \tag{5.1}$$

A constant O denotes an object of the environment, e.g. $Ball$, $OwnGoal$, or players. The function $COND_p$ is specific for each predicate p. For example, the predicate $inReach(O, M)$ is defined as follows (R is the robot itself):

```
COND_inReach(O, M): boolean
    return (dist(R, O) < 1200)
```

$inReach$ is an example for a predicate whose truth value is grounded on the distance between the robot and another object. For convenience, this kind of predicate is called *distance predicate* from now on. Of course, predicates can state various kinds of knowledge about the environment, for example the visibility of objects (e.g. $unknown(Ball)$ is $true$ iff the position of the ball is unknown) or angles between objects.

Based on the current state of the knowledge base, the planner selects a plan which shall be executed next. A plan P comprises:

1. A precondition pre_P which is a conjunction of ground literals. P can be executed only if pre_P is fulfilled.

2. A sequence of actions $[a_1, .., a_n]$. $a_i \in A$ where A is the set of actions the robot is able to execute. A plan is successfully finished iff all actions are finished. The sequence of actions is either dynamically computed using classic AI planning or it is statically defined by a human operator.

3. An invariant inv_P which is a conjunction of ground literals. If an invariant of a currently executed plan is violated, then the plan is aborted.

A more detailed discussion of the plan execution is found in [FSW05].

5.2 A Predicate Hysteresis

The mapping from a quantitative model to symbolic predicates in a dynamic and uncertain environment leads to two major problems: First, the truth value of predicates is calculated using thresholds, i.e., there are sharp boundaries. Thus slight changes of the environment can cause truth value changes and result in abortion of plans due to a violation of the invariant, even if the plan still could be finished successfully. The consequence is instability in the high-level decision making process. A *commitment* to a plan, once it is chosen, is desired. Second, sensor data is inherently noisy. Hence, due to the sharp boundaries, sensor noise leads to *unstable knowledge*, i.e. to undesired oscillation of truth values, even if the environment does not change.

We propose a *predicate hysteresis* as an attempt to mitigate the problems described above. The term *hysteresis* is well known from electrical engineering. It means that the current state is influenced by a decision which has been made previously. We adapt this concept in order to improve the robustness of the decision making process. The basic idea is that, once a predicate

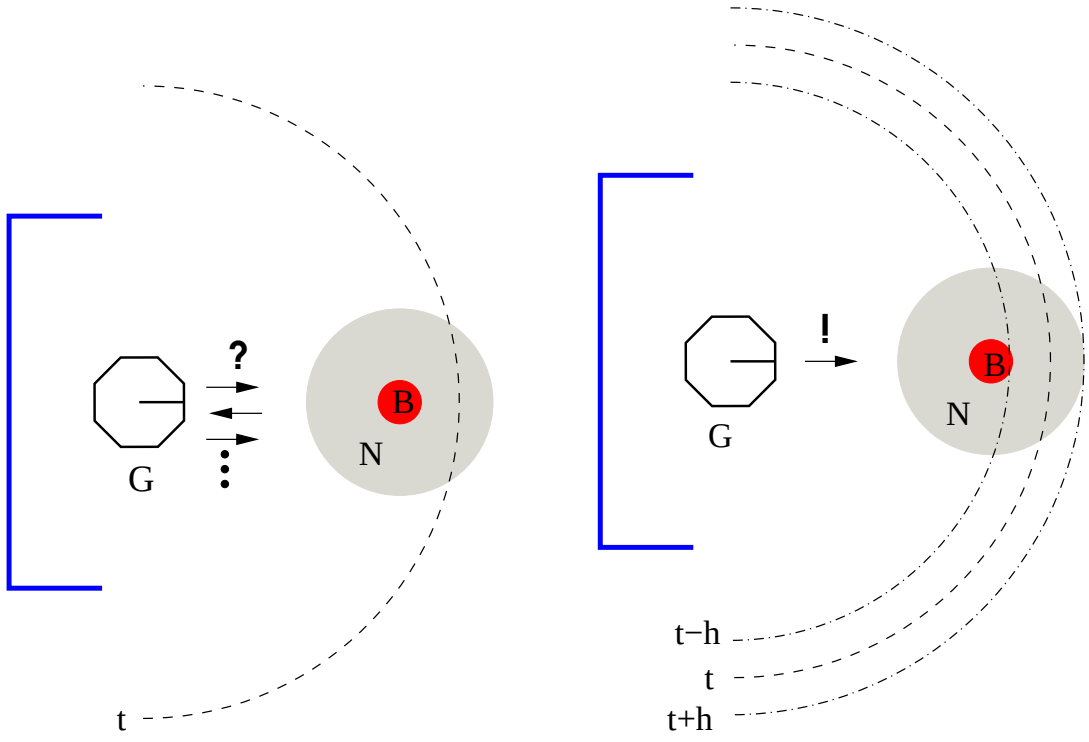

Figure 5.3: Example: (a) no hysteresis, (b) with hysteresis of size *h*. G is the goalkeeper, B the ball, the area N depicts the uncertainty of the ball position measurements.

evaluates to a certain truth value, only significant changes of the environment can cause a change of this truth value.

Thus an extended interpretation function I_H is introduced:

$$I(p(O^n), M, l) = \begin{cases} true & \text{if } COND_p(O^n, M, l) = true \\ false & \text{otherwise} \end{cases} \tag{5.2}$$

The variable l represents the current truth value of p.
The functions $COND_p$ are also redefined. The general definition for distance predicates is:

```
COND_distance_pred(O, M, l): boolean
 if l then
     return (dist(R, O) < th + h
 else
     return (dist(R, O) < th - h
```

th denotes a threshold, which is specific for each predicate. In the example given above, the predicate $inReach$ has a threshold of 1.20m. h is the *hysteresis size*. In this definition, the

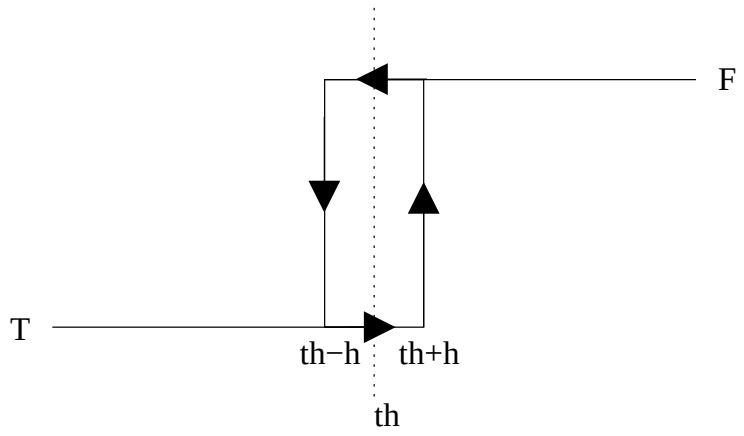

Figure 5.4: Evaluation of the predicate $inReach$ using hysteresis.

hysteresis size is defined as an absolute number. In practice, predicates with a larger threshold may also demand a larger hysteresis because in general the sensor noise is higher for distant objects. Thus it is often more convenient to define a hysteresis size h_{rel} as a percentage of the threshold:

```
COND_distance_pred(O, M, l): boolean
 if l then
     return (dist(R, O) < th * (1+h_rel)
 else
     return (dist(R, O) < th * (1-h_rel)
```

However, in this paper the term *hysteresis size* always denotes an absolute number in mm.

Figure 5.3 gives an example for the effect of using a predicate hysteresis. It shows a goal-keeper in his goal. The ball has approached the goalkeeper and stops at the position which is shown in the figure. $dist(R, Ball)$ is slightly less than $th - h$, whereas th is the threshold of the predicate $inReach$ and h the hysteresis size.

Suppose the goal-keeper's strategy includes the following plans:

	precondition ≡ invariant:	action:
P_1:	$\neg hasBall() \wedge \neg inReach(Ball)$	stay in goal
P_2:	$\neg hasBall() \wedge inReach(Ball)$	grab ball

In (a) as well as in (b), $inReach(Ball)$ becomes $true$ and thus P_2 is activated. The goalkeeper starts moving towards the ball.

But in (a), where no hysteresis is used, it may happen that $inReach(Ball)$ becomes $false$ again due to sensor noise. In this case, the current plan is aborted and P_1 is reactivated. This flipping can happen several times in quick succession, the goalkeeper activates plans and aborts them before they can succeed.

In (b) a hysteresis of size h is used. As soon as $inReach(Ball)$ becomes $true$ (i.e., the distance between the robot and the ball is less than $th - h$), it keeps this truth value as long as the $dist(R, Ball)$ is less than $th + h$. Thus the truth value of $inReach(Ball)$ is, to a certain extent, robust against the noise of the ball position measurement. In this example, the hysteresis size is sufficiently large to compensate the noise, and the goalkeeper does not abort P_2 after he has made this decision. The goalkeeper *commits* himself to this decision — as it would happen in a real soccer match. If a real good goalkeeper leaves his goal in order to grab the ball, he does not change his mind only because of a slight change of the ball position.

5.3 Experimental Results

The proposed symbol grounding with hysteresis was evaluated on our real robots within the robotic soccer domain. We investigated how the use of a hysteresis in symbol grounding stabilizes the evaluation of the truth value of predicates and reduces the number of undesired changes of the truth value caused by noise and changes in the environment.

We conducted several static and dynamic experiments in which the robot measured the distance to objects on the field using its vision system. Based on these measurements, the symbol grounding evaluates the truth value of the distance predicate $inReach$. The distance measurements are not reliable and vary within certain boundaries because of noise and changes in the environmental conditions. Therefore, there are undesired changes in the truth value of predicates even if the distances do not change in the real world.

Figure 5.5 shows series of distance measurements during a static experiment. The robot was placed 4800 mm away from the yellow goal. We recorded series of distance measurements over periods of 30 seconds. These series were recorded at different times during the day to investigate the influence of changing lighting conditions. Please note that the vision system was calibrated the day before and no adaptation of the vision and camera took place between the different series.

As the experiment setup was totally static a perfect vision system would always report the same distance and there would no change in the truth value of a distance predicate. But the Figure shows that in practice the measurements are affected by noise. Furthermore, it shows the clear dependency of the amount of noise in the data on changing lighting conditions. The extent of noise differs within the different series recorded under different lighting conditions. This change is caused by the fact that the color of objects is differently perceived under changing light and the robot vision relies on the colors of objects. The worst conditions were at 17:00 where it became

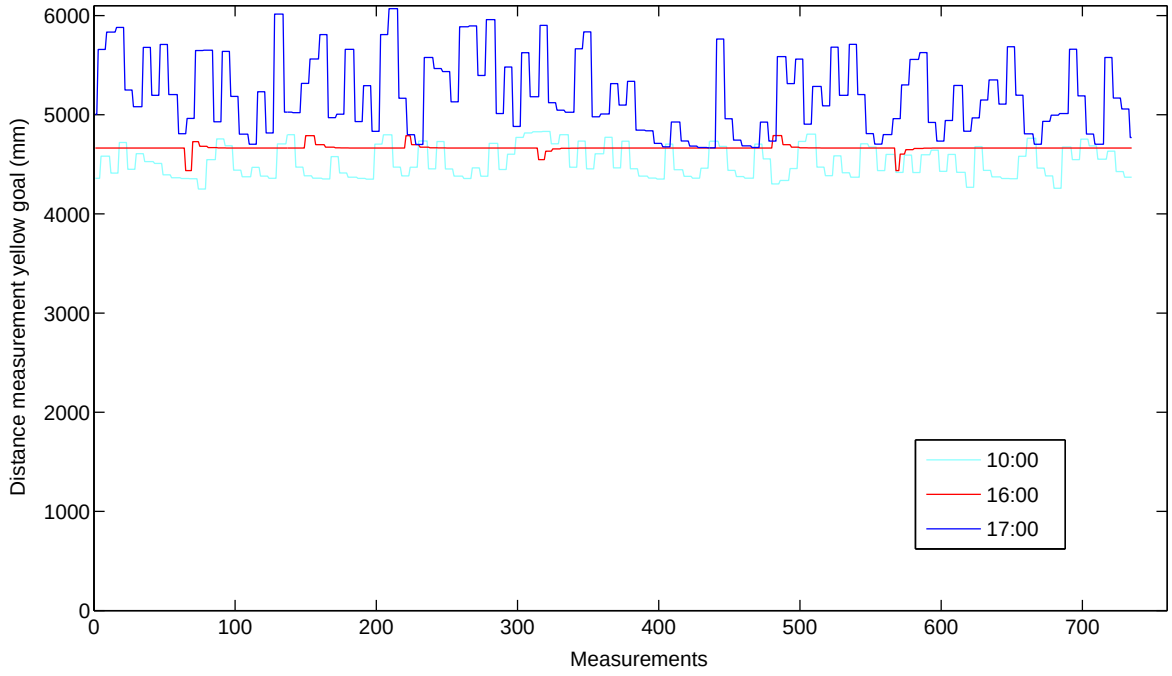

Figure 5.5: Distance measurements for a static object 4800 mm away from the robot at different times during a day.

dark.

# Meas.	μ	σ	Δ	n			
				h=0	h=σ	h=$\frac{\Delta}{3}$	h=$\frac{\Delta}{2}$
	mm	mm	mm				
735	5202	385	1403	61	17	11	0

Table 5.1: Number of undesired truth value changes n of predicate $inReach$ for the yellow goal for static distance measurements at 17:00 with different sizes h for the hysteresis.

Because of the quality of the distance measurements, the symbol grounding with a fixed threshold reports a number of truth value changes of the distance predicate. These changes are undesired because the object positions did not change in the real world. Table 5.1 shows the results of the symbol grounding of the series at 17:00 and reports how different sizes of a hysteresis stabilize the symbol grounding. The series contained 735 measurements with a distance mean of 5202 mm and a standard deviation σ of 385 mm. The value Δ is the difference

between the maximum and the minimum of the measured distance within this series. If we do not use a hysteresis in the symbol grounding, we get 61 undesired changes of the truth value. If we use a hysteresis with the size of σ then we reduce the number of changes to 17. An increase of the size of the hysteresis to $\Delta/2$ reduces the changes to zero. This clearly shows the benefit of the use of a hysteresis. But the size of the hysteresis is always a trade-off between stability and reactivity of the system. One has to take care that the size of the hysteresis does not exceed an adequate level. Otherwise, the system will lose its reactivity.

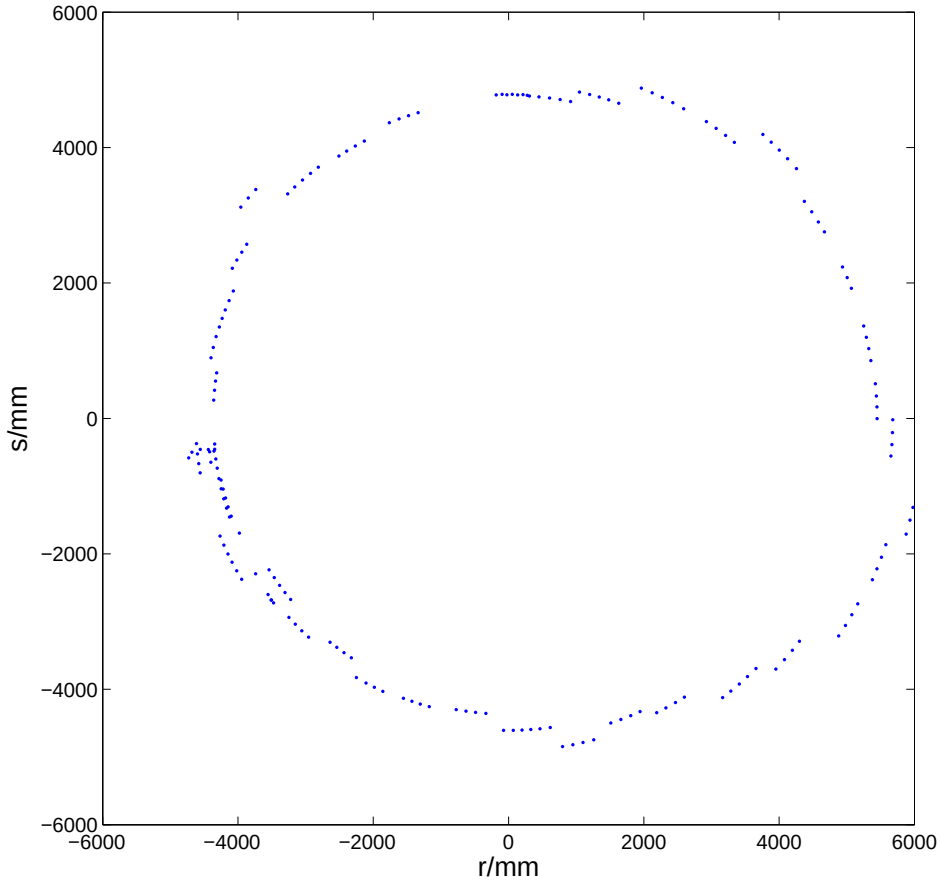

Figure 5.6: Position measurement for a static object 4000 mm away from the robot while the robot rotates. Positions are shown in the robots local coordinate system.

Figure 5.6 shows position measurements for an object within a dynamic experiment. The robot was placed 4000 mm away from the yellow goal. We recorded a series of position measurements while the robot performed a full rotation around its vertical axis. Positions are shown in the robots local coordinate system. The robot is located in the origin and the positive r-axis

points to the front of the robot. If there were no inaccuracy in the vision system and the motion of the robot then the position measurements would lie on a perfect circle and the distance measurements to the object would remain constant. But the real measurements are affected by errors. There are three major reasons for these errors. First there is noise from the vision system. Furthermore, there is inaccuracy in the tracking of the object with the Kalman filter. This effect causes the tangential drift and the discontinuity of the measurements. Finally, the imperfect geometric calibration of the camera causes a deformation of the hypothetic circle. Distances to objects in the rear appear shorter in the camera as distances to objects in the front.

# Meas.	μ	σ	Δ	n			
				h=0	h=σ	h=$\frac{\Delta}{3}$	h=$\frac{\Delta}{2}$
	mm	mm	mm				
329	4789	485	1937	4	3	3	1

Table 5.2: Number of undesired change n of predicate $inReach$ for the yellow goal for rotating distance measurements with different sizes h for the hysteresis.

Table 5.2 shows the evaluation of the above position measurements. The position measurements were converted to distances by calculating the Euclidean distance. Without using a hysteresis there are four undesired changes in the truth value of the $inReach$ predicate for the yellow goal. The table shows that with an increasing size of the hysteresis the number of undesired changes decreases to one. Please note that because all predicates are initialized with false there is always one change in the truth value even if the predicate is always correctly evaluated true.

Figure 5.7 shows the results of another dynamic experiment. In the experiment the robot was placed 4000 mm away from the ball and directly facing it. We recorded the distance measurements to the ball while the robot was directly approaching it. If we assume again a perfect perception then distances are supposed to monotonically decrease. The figure clearly shows that this is not the case in our real experiment due to the imperfect perception. The evaluation of the symbol grounding without hysteresis for this experiment reports three undesired changes of the truth value of the predicate $inReach$ for the ball. We calculated the mean and standard deviation of the differences of succeeding distance measurements, but we only considered cases in which the measured distance increased. The mean was 51 mm and the standard deviation was 8.8 mm. If we used a hysteresis with a size of σ, there were no undesired changes anymore.

The results of the above experiments in the real world show that quantitative perception is always affected by noise, changes in the environment and other inaccuracies. Therefore symbol grounding with simple thresholds can not be stable even the world does not change. This instability negatively affects the performance of the qualitative planning and reasoning process of an

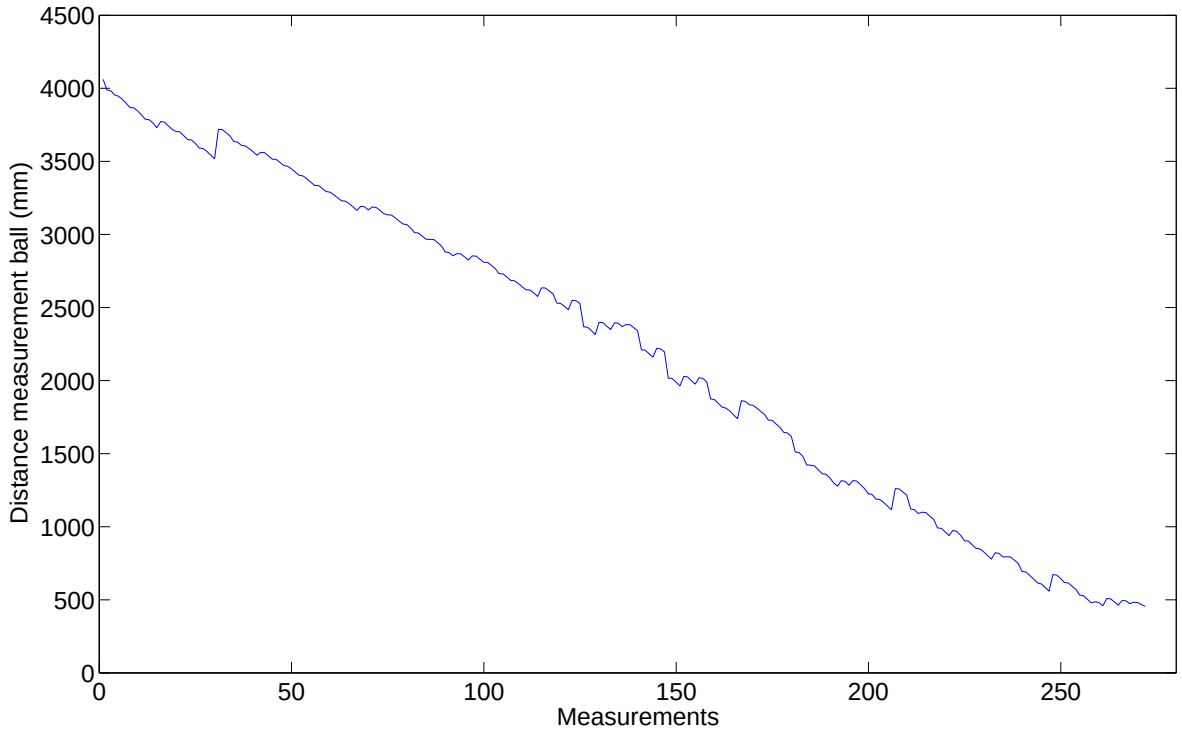

Figure 5.7: A sequence of consecutive distance measurement for a static object while the robot directly approaches the object.

agent. Furthermore, the results show that the proposed symbol grounding with hysteresis is able to decrease the number of undesired changes of truth values of predicates to a minimum. This leads to an improvement of the stability of the robot's knowledge and increases the performance of the qualitative decision making process.

5.4 Open Issues

Although the use of a hysteresis in symbol grounding is justified by experimental results, there are still open questions concerning the proposed method.

There is no general answer to the question how the size of the hysteresis can be satisfactorily chosen. The size which is appropriate to sufficiently stabilize the symbol grounding while keeping the system reactive may differ from situation to situation. For example, the light conditions are always different and unpredictable. Furthermore, a more careful investigation should be done on the impact of the hysteresis on the reactivity of the system. A open question in this context is the definition of an appropriate evaluation criteria. A quick idea might be to play a dozed

simulated games with and without the hysteresis and to compare the results, like goals scored or games won.

So far, we have not done any quantitative evaluation of how the symbol grounding with hysteresis influences the planning and plan execution in situations with slightly changes in the world. We assume that the hysteresis increases the performance of the plan execution as it can be compared to a commitment to follow a certain plan even if there are changes in the environment.

More research should be done on the conjunctions of predicates using hysteresis. Assume we use a conjunction of a large number of these predicates. If all measurements for predicates reach the upper boundary of their hysteresis the qualitative situation is the same as all measurements for predicates lie around the lower boundary. But the quantitative situations in the real world may substantially differ.

We use some predicates in different plans. Regardless of in which plan a predicate is used we use the same hysteresis size for the predicate. It might be desirable to use different hysteresis sizes for the same predicate in different situations in order to adjust the stability and reactivity of a predicate for a certain situation.

A small size for the hysteresis eliminates instabilities in the truth value without a significant decrease of the reactivity of the predicate. We need an even larger hysteresis if the inaccuracies in the perception become larger. But this fact negatively affects the reactivity of the system. It might be interesting if a smaller hysteresis is sufficient if more qualitative knowledge about how the world works is added to the reasoning.

5.5 Related Work and Discussion

In [DFL03] and [PAC04] hybrid systems for controlling robots of a RoboCup MSL Team were presented. Both use *Golog* for the representation of the qualitative model and the derivation of plans. Furthermore, they use the qualitative model and decision trees to evaluate the most appropriate action in a certain situation. This action is used if no plan is available in that situation. The used action might not be the best but keeps the robot reactive even the planning take some time. However, the problems arising from noise and jitter in the quantitative model were not accounted in those approaches. In [Ree99], an approach to action selection in Robotic Soccer is presented. The *action modules*, which are introduced in this work, have preconditions and invariants. The invariants can contain fewer conditions than the related preconditions in order to avoid oscillating behavior. The modules also have *activation factors* stating the utility of the action. These factors are situation independent, but are increased during the execution of an action module. This results in larger robustness of the behavior. In the work presented in [Mül00] a similar approach was used for gaining robustness. Sachenbacher and Struss [SS01] presents a framework for automated qualitative abstraction of quantitative models. However, a

complete knowledge of the quantitative model is required, whereas in our case only quantitative observations (which are incomplete and uncertain) are mapped. There are approaches which avoid the addressed problems in symbol grounding by the usage of reasoning with uncertainties like fuzzy logic or probabilistic networks. However, these approaches require different models and modeling processes.

In this paper we addressed the problem of symbol grounding in applications with a very high degree of (mostly unpredictable) interactions. We introduced the concept of predicate hysteresis to overcome some of the corresponding problems that occur in practice. We further described empirical results we obtained when using predicate hysteresis for symbol grounding on our robots. The outcome of the predicate hysteresis substantially improved the behavior of the robots. We further discussed open issues and future research directions.

Chapter 6

Model-Based Diagnosis for Robot Control Software

Control software of autonomous and mobile robots is characterized by its fairly high complexity which is in conflict with runtime requirements like stability and flexibility. Complexity is caused by the software components implementing the basic functionality like planning, computing world models, sensor and actuator interfaces, and their connections. Because of the high complexity, the instability of hardware components and connections, and the underlying operating system, complete stability of such a platform is very unlikely. This problem description does not only hold for mobile robots but all systems comprising software and hardware which interact with the real world. But when we want to build a robot that is truly autonomous, it has to deal with failures during its runtime without degrading the desired behavior or even worse failing to fulfill its mission. Hence, a diagnosis system on top of the control system which does monitoring the current behavior, locating the cause of a detected failure, and taking the appropriate actions is a necessity. In this chapter we will present a model-based framework for this purpose. Parts of the framework were also published in [SW05b, SMW05, SW05c].

There are several requirements for a diagnosis system in the domain of mobile robots. First, ideally the diagnosis system should not cause any changes of the control systems. If changes are necessary, they should be as small as possible. Furthermore, the diagnosis system must not affect the behavior of the control system. This requirement is very important in order to keep the effort for introducing a diagnosis system as small as possible. Second, the diagnosis system should not reduce the overall available computational power because this might decrease the robot's functionality, e.g., it's ability to react to a given event in a certain amount of time. Third, in cases where the diagnosis system itself does fail, there should be (almost) no effects on the control system. Finally, the memory requirements of the diagnosis system should be as small as possible. Otherwise, the diagnosis system has a too large effect on the system performance. For

critical application it is desirable that the integration of a diagnosis system is part of the initial design.

In order to fulfill the above requirements we introduce a model-based solution. This includes a model of software components and their relationships that are specified in the software architecture of the robot's control system. This model is then used to derive root causes of a detected failure. A root cause itself is a software component. The failure detection is based on observations. For this purpose we use the concept of observers, i.e., software programs that monitor system activities like the number of active processes of a software component. If the monitored value exceeds its pre-specified boundaries, the observer raises a conflict which causes the diagnosis engine to compute the root causes. Once the root cause has been identified, the diagnosis system takes appropriate actions in order to retain the system's correct behavior. Possible actions are killing and re-generating processes that caused the failure. The fault detection, localization, and correction procedures are all based on declarative models of the control software.

In this chapter, we present the foundations of model-based diagnosis, the modeling paradigms, the observers, and the algorithm for retaining the correct state. Moreover, we present the results of a case study which had been realized by modeling the control software architecture of our mobile robot. In the used test-cases typical failures, like software components that become inactive because of deadlocks, are represented. The case study shows that the overall system performance is not degraded and that the diagnosis system always retains the desired state.

6.1 Model-based Diagnosis

6.1.1 Foundations

In the previous section, we briefly outlined how model-based diagnosis works and how it can be used to detect faults in the control software of autonomous mobile robots. In this section we will provide a deeper knowledge about the foundations of model-based diagnosis.

An overview on the process of model-based diagnosis is shown in Figure 6.1. The fundamental principle of model-based diagnosis [Rei87] is that it uses a description of the correct behavior of components and the connections among the components to detect and locate faults in a system.

The model of the correct behavior of a system, the system description, is derived from the specifications or requirement of the system. Such requirements or specifications are usually present or easily to obtain. For the representation of the system description a logic-based form is preferred in order to ease deduction and reasoning. This model does a prediction of the correct output of the system based on current inputs. The advantage of the use of this kind of system

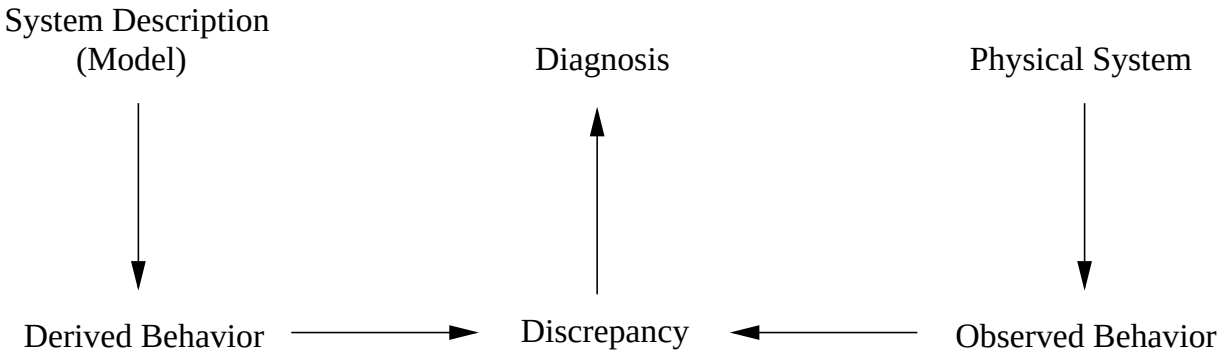

Figure 6.1: Overview of the diagnosis process.

description for diagnosis is that only the correct behavior has to be modeled and theoretically all faults can be detected. In other techniques for fault detection and identification (FDI) [VGST04] each possible fault which one wants to detect has to be explicitly modeled.

The current outputs of the system are represent by observations. Such observations simply are a snapshot of the current system behavior. Such observation range from simple discrete information, e.g, input pins of digital circuit, up to advanced temporal integration of continuous values, e.g., multi hypotheses tracking [LBS96]. Again, a logic-based representation of the observation is desirable.

If a contradiction between the output of the system description (desired or correct behavior) and the observations (current behavior) exist we have detected a fault in the system. This is the fact because only a faulty system will show a different behavior than the expected correct behavior. Detection of the contradiction can be easily done by logic inference if the system description and the observation available in a logic-based representation.

Once a fault has been detected the interesting question remains which is the root cause for the fault. In model-based diagnosis the localization of the root cause is done by a systematic attempt to resolve the contradiction. Such a resolving is based on adding and removing of assumptions about faulty components. We use Reiter's algorithm [Rei87] in combination with a fast propositional theorem prover [Min88] for this purpose.

More formally, the parts of the model-based diagnosis can be defined in the following way:

Definition 6 (Diagnosis System) *A diagnosis system is the tuple (SD,COMP) with:*

1. *the system description (SD) is a set of logical sentences (horn clauses or first order logic) that specifies the correct behavior of the components and the connections among the components*

2. *COMP is the set of components*

It forms together with current observation of the system OBS the diagnosis problem.

Definition 7 (Diagnosis Problem) *A diagnosis problem is the triple (SD,COMP,OBS) with:*

1. *the system description (SD) is a set of logical sentences (e.g., horn clauses) that specifies the correct behavior of the components and the connections among the components*
2. *COMP is the set of components*
3. *OBS is a set of system observations*

Definition 8 (Diagnosis) *A diagnosis of a diagnosis problem (SD,COMP,OBS) is a set* $\Delta \subseteq$ *COMP so that SD* $\cup$ *OBS* $\cup \{\neg$ *AB(C)* $\mid$ *CO* $\in$ *COMP* $\setminus \Delta\} \cup \{$ *AB(CO)* $\mid$ *CO* $\in \Delta\}$ *is consistent. AB(CO) states that the component CO shows an abnormal behavior. A diagnosis is minimal iff no proper subset is a diagnosis.*

Intuitively, a diagnosis Δ is a set of faulty components which explains inconsistency between the desired and the observed behavior of a system. In general, in practical applications one prefers the derivation of minimal diagnosis. Naively, all diagnosis Δ can be computed by simple checking the above properties for all possible subsets of *COMP*. But such an approach is not feasible in practice because even for small systems the computational costs explode. Therefore, Reiter [Rei87] proposed a more efficient and elegant way to compute diagnosis. He used conflicts and hitting sets.

Definition 9 (Conflict) *A conflict set of a diagnosis problem (SD,COMP,OBS) is a set CO* $\subseteq$ *COMP such that SD* $\cup$ *OBS* $\cup \{\neg$ *AB(CO)*$\mid$*CO* $\in$ *COMP*$\}$ *is inconsistent. A conflict set is minimal iff no proper subset is a conflict set.*

The conflict covers simply the property that if all components in the conflict are assumed to work correct this leads to a inconsistency. Therefore, one has to declare at least one component in the conflict set malfunctioning in order to resolve the conflict. If one is able to find a collection of components assumed to be malfunctioning which resolves all conflicts a diagnosis is found. Such calculations can be performed efficiently by using hitting sets.

Definition 10 (Hitting Set) *Assume* C *is a collection of sets. A hitting set for C is a set* $H \subseteq \bigcup_{S \in C} S$ *such that* $H \cap S \neq \emptyset$ *for each* $S \in C$*. A hitting set is minimal iff no proper subset is a hitting set.*

Hitting sets are useful in the calculation of diagnosis because of the following theorem.

Theorem 4 *A set Δ is a (minimal) diagnosis for the diagnosis problem (SD,COMP,OBS) iff Δ is a (minimal) hitting set for the collection of conflicts sets.*

An algorithm for the efficient calculation of hitting sets and diagnosis will be outlined later in this chapter.

6.1.2 Simple Example

In this section we will explain the fundamental principles of model-based diagnosis using a simple example. We will deploy model-based diagnosis on the simple circuit shown in Figure 6.2. The circuit comprises three multipliers $\{M1, M2, M3\}$ and two adders $\{A1, A2\}$. Therefore, the set of components $COMP$ is $\{M1, M2, M3, A1, A2\}$. The circuit performs a simple arithmetic calculation. Figure 6.2 shows the structure of the circuit and the observed inputs and outputs (green numbers). As one can simply follow, the output of the circuit is in contradiction with the expected outcome (red numbers). The output f should have the value 12. But the observed value at terminal f is 10. In order to show the principles of model-based diagnosis, we will apply it to this example to detect and locate faulty components.

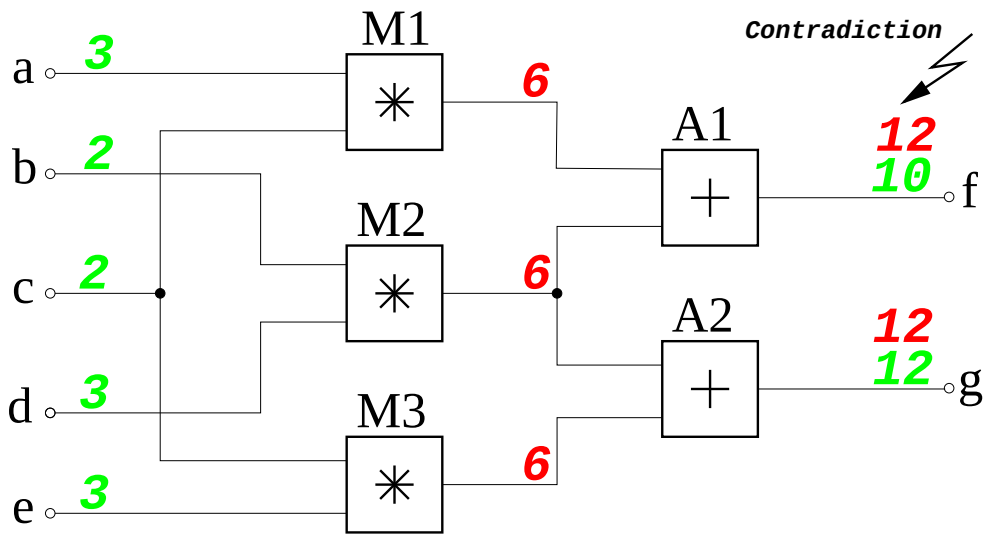

Figure 6.2: Simple diagnosis example with contradiction between the modeled behavior and the observations. Observations of the system are shown in green. Predictions from the system model are shown in red.

First we have to build up the system description *SD* which comprises models of the desired behavior of the components and a description of the connections between the component.

The behavior model of a multiplier m can be described with the following logical sentences:

1. $\neg AB(m) \wedge Mul(m) \wedge in_1(m) = x \wedge in_2 = y \rightarrow out(m) = x * y$
2. $\neg AB(m) \wedge Mul(m) \wedge out(m) = x \wedge in_1 = y \wedge in_1 \neq 0 \rightarrow in_2(m) = x/y$
3. $\neg AB(m) \wedge Mul(m) \wedge out(m) = x \wedge in_2 = y \wedge in_2 \neq 0 \rightarrow in_1(m) = x/y$
4. $\neg AB(m) \wedge Mul(m) \wedge out(m) = 0 \wedge in_1(m) \neq 0 \rightarrow in_2(m) = 0$
5. $\neg AB(m) \wedge Mul(m) \wedge out(m) = 0 \wedge in_2(m) \neq 0 \rightarrow in_1(m) = 0$

Line 1. specifies the behavior in forward direction. The line states that the output of a multiplier simply is the product of its inputs. The lines 2. to 5. allow a backward reasoning. If one knows the value of the output and one input the value of the second input can be calculated. The predicate $Mul(m)$ indicates that the component m is a Multiplier. This predicate is later used to build up the description of the complete system.

The behavior model of an adder a can be described with the following logical sentences:

1. $\neg AB(a) \wedge Add(a) \wedge in_1(a) = x \wedge in_2(a) = y \rightarrow out(a) = x + y$
2. $\neg AB(a) \wedge Add(a) \wedge out(a) = x \wedge in_1(a) = y \rightarrow in_2(a) = x - y$
3. $\neg AB(a) \wedge Add(a) \wedge out(a) = x \wedge in_2(a) = y \rightarrow in_1(a) = x - y$

Line 1 specifies the behavior in forward direction. The line states that the output of an adder simply is the sum of its inputs. The lines 2 to 3 allow a backward reasoning. The predicate $Add(a)$ indicates that the component a is an Adder.

The structure of the system and the connections among the components can be specified as the following logical sentences which are represent a collection of facts:

1. $\Rightarrow in_1(M1) = a$
2. $\Rightarrow in_2(M1) = c$
3. $\Rightarrow in_1(M2) = b$
4. $\Rightarrow in_2(M2) = d$
5. $\Rightarrow in_1(M3) = c$

6. $\Rightarrow in_2(M3) = e$
7. $\Rightarrow in_1(A1) = out(M1)$
8. $\Rightarrow in_2(A1) = out(M2)$
9. $\Rightarrow in_1(A2) = out(M2)$
10. $\Rightarrow in_2(A2) = out(M3)$
11. $\Rightarrow in_2(M1) = in_1(M3)$
12. $\Rightarrow in_2(A1) = in_1(A2)$
13. $\Rightarrow f = out(A1)$
14. $\Rightarrow g = out(A2)$

The lines 1. to 6 specify the connections for the input terminals. The lines 7. to 12. specify the connections among the components. The lines 13. to 14. are responsible for the connections to the output terminals.

Finally, the following fact define the type of the different components.

1. $\Rightarrow Mul(M1)$
2. $\Rightarrow Mul(M2)$
3. $\Rightarrow Mul(M3)$
4. $\Rightarrow Add(A1)$
5. $\Rightarrow Add(A2)$

Up to now we have the system description *SD* which tells us how the system has to perform if all components work properly. The actual behavior of the system is monitored by a set of observations *OBS*. The observations of the system are the values of the input and output terminals. It has to be noted that there is a wide range of different types of observations in different applications. The values of observations are also specified as logical facts.

1. $\Rightarrow a = 3$
2. $\Rightarrow b = 2$

3. $\Rightarrow c = 2$
4. $\Rightarrow d = 3$
5. $\Rightarrow e = 3$
6. $\Rightarrow f = 10$
7. $\Rightarrow g = 12$

The lines 1. to 5. specify the observed values of the input terminals. The lines 6. to 7. specify the observed values of the output terminals.

The next step to deploy the system description *SD* to deduct the values the system has to calculate if every component of the system works correct. These deductions (the red numbers in Figure 6.2) are $\{f = 12, g = 12\}$. Sometimes simple forward propagation of the values is applied. Obviously, these results are in contradiction with the observations. This contradiction shows that our assumption that all components work correctly is not true. Therefore, we have detected a fault in the circuit. Such a consistency check are usually performed by using a theorem prover, e.g., [Min88].

In our example we have two minimal conflicts. The minimal conflict sets $\{M1, M2, A1\}$ and $\{M1, A1, M3, A2\}$ are shown in Figure 6.3.

From the intersection of the two minimal conflict sets $\{M1, A1\}$ we can derive two minimum hitting sets $\{M1\}$ and $\{A1\}$. Therefore, $AB(M1)$ and $AB(A1)$ are single fault diagnoses. This is obvious because each of the two diagnosis resolves both conflicts. Furthermore, the sets $\{M2, A2\}$ and $\{M2, M3\}$ also resolve both conflicts. Therefore, we have derived two multiple fault diagnoses.

6.2 Modeling Software Architectures

Software architectures provide a general view on software. Software architectures comprise software components and their connections. Components represent a collection of classes which implement a certain behavior. The connections between components represent dependency relations like client-server relationships and data flow. For example a robot's architecture might comprise components for image processing, motion control, planning actions, and others. During the execution of a program the components might spawn processes and interact using method calls, or other means of communication, like events. Figure 6.4 depicts parts of the software architecture of our mobile robot.

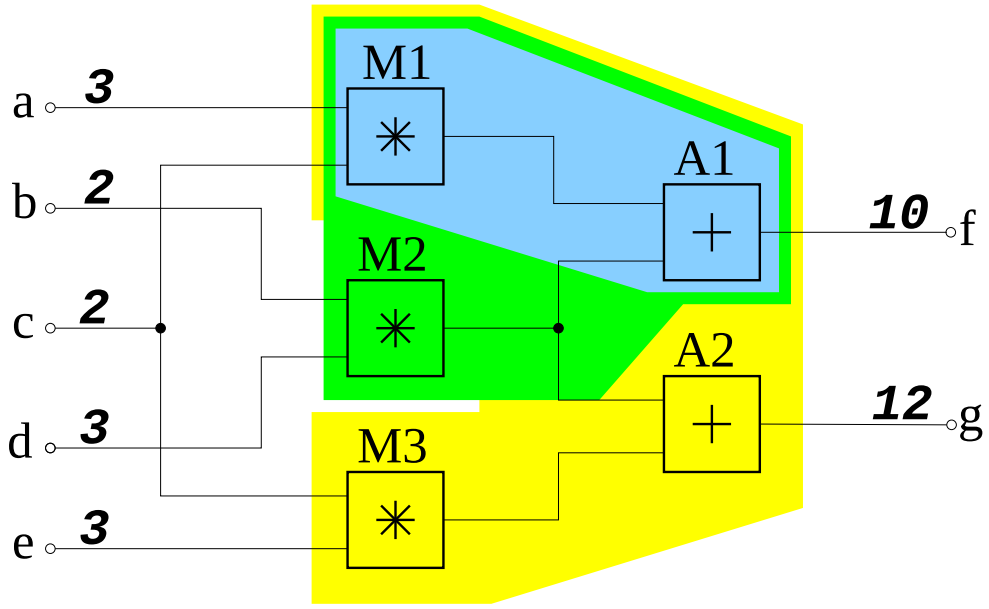

Figure 6.3: Simple diagnosis example with the minimal conflicts. The first minimal conflict comprises the components $\{M1, M2, A1\}$ and is shown in green. The second minimal conflict comprises the components $\{M1, A1, M3, A2\}$ and is shown in yellow. The intersection of the booth conflicts $\{M1, A1\}$ is shown in blue.

The following formalization of the structural properties of the software architecture considers the software components, their connections in terms of identifiers representing events or procedure calls, and a classification of dependency relations which are used to repair the software during runtime. We distinguish two different dependency relations between components, namely weak and strong dependencies. Two components are weakly dependent if killing one component at runtime does not require the other component to be killed and to be re-started in order to repair the overall system. Otherwise, the relationship is a strong dependency.

Definition 11 (SAM) *A software architecture model (SAM) is a tuple (CO,C,out,in,WDC,SDC) where:*

- *a set of software components CO*
- *a set of connections C*
- *a function out: CO* $\mapsto 2^C$ *returning the output connections for a given component*
- *a function in: CO* $\times C \mapsto 2^C$ *returning the input connections for a given component and an output connection. This function only returns those inputs that influence the value of the specified output.*2^C *demotes the power set of the connections* C*.*

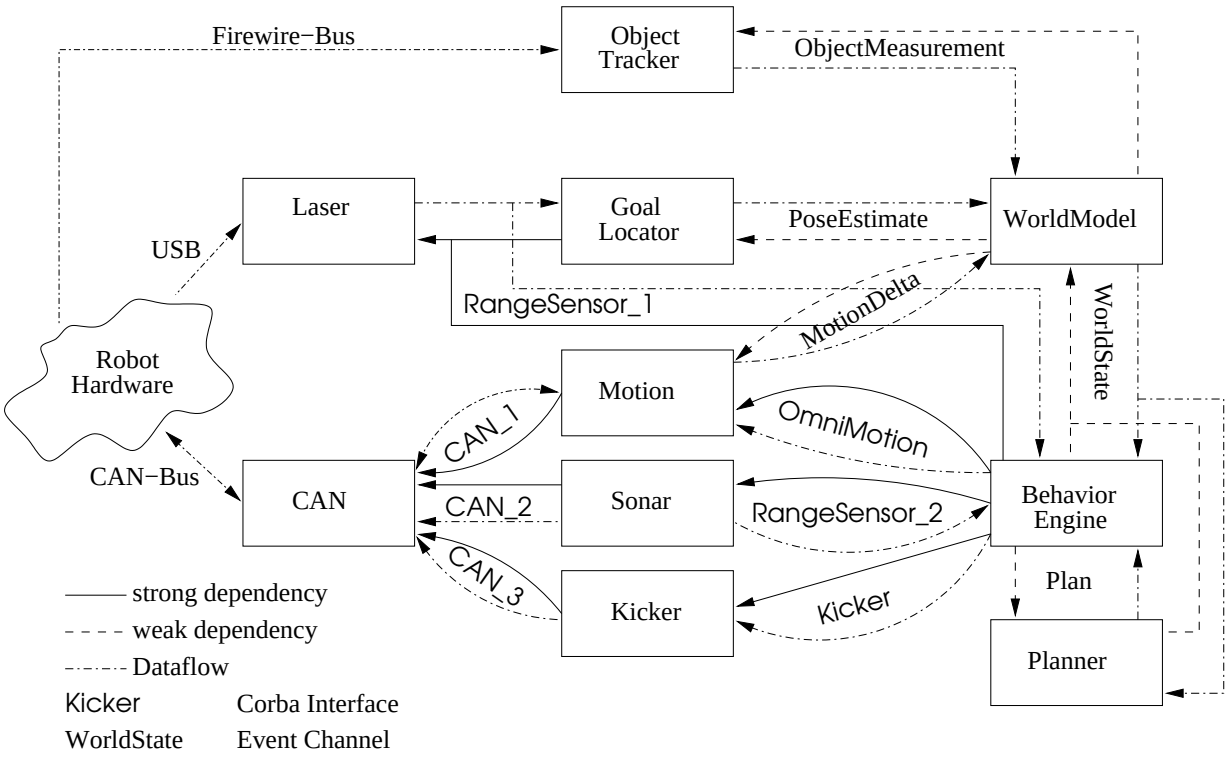

Figure 6.4: Dependencies between software and hardware modules

- *a set of weak dependencies WDC* $\subseteq 2^{CO \times CO}$
- *a set of strong dependencies SDC* $\subseteq 2^{CO \times CO}$

We represent all weak and strong dependencies as ordered pair (x_1, x_2) *with* $x_1, x_2 \in$ *CO. The direction of the connection is from* x_1 *to* x_2*.*

Corollary 5 *If* $(x, y) \in$ *SDC and* $(y, z) \in$ *SDC then* $(x, z) \in$ *SDC holds.*

These reflexive relation is later used to derive appropriate repair actions.

Hence, the SAM representing the software architecture of Figure 6.4 is :

$(\{LASER, CAN, OT, GL, MO, SO, KI, WM, BE, PL\},$
$\{ObjectMeasurement, WorldState, \ldots\},$
$\{out(MO) = \{MotionDelta\}, \ldots\},$
$\{in(MO, MotionDelta) = \{CAN\_1\}, \ldots\},$
$\{(WM, OT), ..\}, \{(LASER, GL), \ldots\})$

where OT is the object tracker, GL the goal locator, MO the motion service, SO the sonar service, KI the kicker service, WM the world model, BE the behavior engine, and PL the planner.

The concrete behavior of the software at runtime is determined by the implemented behavior of its software components. A formalization of the concrete behavior requires the transformation of the whole program which is not only a very difficult task but leads to models that can hardly be used for diagnosis at runtime where resources for diagnosis are limited. Hence, an abstraction of the concrete behavior is necessary. The idea behind the abstract behavior model of software components is similar to models which are based on dependencies like the one described by [FSW99]. If all inputs to the model are correct, a software component should produce a correct output. This conversion has to be performed for all components and their output connections.

The algorithm for performing this conversion is as follows where the predicate AB stands for abnormal, and ok indicates a correct event or method call.

computeModel(CO, C, out, in, WDC, SDC**)**
Input: the SAM.
Output: a set of horn clauses

1. Let M be the empty set.
2. For all $x \in CO$:
   (a) Add $\neg AB(x) \rightarrow ok\_num\_processes(x)$ to M.
   (b) For all $e \in out(x)$ add

$$\neg AB(x) \wedge \bigwedge_{e' \in in(x,e)} ok(e') \rightarrow ok(e)$$

   to M.
3. Return M.

Line 2.(a) introduces a rule which says that a correct component spawns the correct number of processes at runtime. Because this parameter of a software component can be easily checked via operating-system calls, we incorporate this knowledge in our model.

For example, the rule that represents the abstract behavior of the OT (Object Tracker) component is:

$$\neg AB(OT) \wedge ok(Firewire) \rightarrow ok(ObjectMeasurement)$$

The size of the model in terms of number of literals depends on the number of the components, the maximum fan-in, and the maximum fan-out of the components. The fan-in and the fan-out are both bound by the cardinality of the connections.

Theorem 6 *The number of literals occurring in the model returned by calling computeModel(CO,C,out,in,WDC,SDC) is* $O(|CO| \cdot |C|^2)$.

If the maximum fan-in and the maximum fan-out are much smaller than the number of components, the number of literals is of order $O(|CO|)$ which is almost always the case for practical applications.

In order to locate root causes, i.e., the components of the software architecture which cause a detected misbehavior, we have to introduce a notation of observations at the same conceptual level. The easiest way of doing this (which has also been done by [FSW99]) is to use the same ok predicate for the purpose. If for example we detect a misbehavior at $ObjectMeasurement$, we could represent this by the literal $\neg ok(ObjectMeasurement)$. The drawback of this representation is the impossibility of distinguishing observations and computed values. Hence, it would be better to introduce a distinguished predicate $correct$ for observations.

Definition 12 (Observation) *Given a SAM* (CO, C, CS). *Either* $correct(x)$ *or* $\neg correct(x)$ *are observations for a connection* x. *The predicate* $correct(x)$ *is true whenever the observed connection shows the correct behavior. If we observe a failure for* x, *the observation has to be* $\neg correct(x)$.

The final step for generating the model is to add rules for coupling observations to models generated by $computeModel$. The following algorithm provides this information.

computeOBSModel(CO, C, out, in, WDC, SDC**)**
Input: The SAM.
Output: A set of horn clauses representing the interface between the architecture model and observations.

1. Let M be the empty set.
2. For all $e \in C$ add
   $correct(e) \rightarrow ok(e)$ and $\neg correct(e) \rightarrow \neg ok(e)$
   to the model M.
3. For all $x \in CO$ add
   $$\bigwedge_{e' \in out(x)} correct(e') \rightarrow \neg AB(x)$$
   to the model M.

4. For all $x \in CO$ add

$$correct\_processes(x) \rightarrow ok\_num\_processes(x)$$

to the model M.

5. Return M.

Line 2. of $computeOBSModel$ provides the interface between the observations and the derived values. Line 3. captures a case where everything that is computed by a software component is known to be correct. In this case it is very likely that the component itself is correct which is represented by third line. In line 4. we provide an interface to the number of processes counter because every component spawns a fixed number of processes. Because the number of processes for each component is only a necessary condition for the correctness of a component it is not correct when saying that the correct number of processes implies the correctness of the component. Therefore, we do not add such a rule to our model.

For our example $computeOBSModel$ would return the following rules for OT and $ObjectMeasurement$:

$$\begin{aligned} correct(ObjectMeasurement) &\rightarrow \\ &ok(ObjectMeasurement) \\ \neg correct(ObjectMeasurement) &\rightarrow \\ &\neg ok(ObjectMeasurement) \\ correct(ObjectMeasurement) &\rightarrow \neg AB(OT) \end{aligned}$$

Theorem 7 *The number of literals occurring in the interface model returned by calling* $computeOBSModel((CO, C, out, in, WDC, SDC))$ *is* $O(|C| \cdot |CO|)$.

6.3 Monitoring Events, Method Calls, and Processes

Coupling the running program with its software architecture model requires an abstraction step. The running program changes its state via changing variable values which is caused by inputs from the environment. This state change is not represented in the SAM. Instead SAM represents the software components and their communication means. Therefore, we require to map changes to communication patterns. For this purpose we introduce the concept of observers. An observer is a piece of software that monitors a certain part of the program's behavior during the execution. For example, an observer might check whether the number of processes for one software component is equivalent to the specified one. Or an observer checks whether a software component produces a number of events during a certain amount of time. If an observer detects

a behavior that contradicts its specification, it computes the appropriate observations in terms of setting the observation predicates $\neg correct(x)$ for the corresponding connection x, and invokes the diagnosis engine.

Definition 13 (Observer) *An observer is a tuple* (S, Ω) *with:*

1. *a set of rules* S *which provides the specification for testing the behavior.*
2. *a set* Ω *comprising predicates which correspond to the observations for the SAM model.*

The specification of the observer determines its abilities of detecting a misbehavior. An observer for checking the number of processes of a given software component specifies exactly this number. In the current implementation of the observer module we allow to specify the following observers:

- *Periodic event production*: This rule is of the form `produces` e `every` n `ms` and checks whether an event e is produced at least every n milliseconds.
- *Conditional event production*: This rule is of the form `produces` e_1 `ever` n `ms after` e_2 and checks whether an event e_1 is produced at least every n milliseconds after the occurrence of an event e_2.
- *Spawn processes*: This rule checks whether a component spawns a number n of named processes id and is of the form `spawns` n `processes` id.
- Periodic method calls: This rule is for checking whether a component calls a method m at least every n milliseconds. `calls` m `every` n `ms`

The observers are used to monitor the state of the system. For this purpose the observers are implemented and check their rules on a regular basis. In cases of failure the diagnosis procedure is invoked. The following algorithm specifies the monitoring process that has been implemented in our system.

monitoring($(CO, C, out, in, WDC, SDC), OS$**)**
Input: The SAM and a set of observer OS.

1. $M_S = computeModel((CO, C, out, in, WDC, SDC))$.
2. $M_O =$
   $computeOBSModel((CO, C, out, in, WDC, SDC))$.
3. $M = M_S \cup M_O$.

4. Do forever:

   (a) Let *OBS* be the empty set.

   (b) For all $os \in OS$ do:

      i. Check the observer os.

      ii. If $os = (S, \Omega)$ detects a misbehavior, add $\bigwedge_{o \in \Omega} \neg o$ to *OBS*.

      iii. Otherwise, add $\bigwedge_{o \in \Omega} o$ to *OBS*.

   (c) If at least one observer detects a failure, call the diagnosis procedure using the model M and the observations *OBS*.

Because of the simplicity of the rules monitoring does not take a lot of time. In case of a failure of course the diagnosis procedure has to be invoked which is more time demanding. However, in this case the system is not in a correct state and resources are necessary in order to reset the system. The implementation of the observers sometimes require additional annotations within the original program. In cases where the communication between components is implemented using for example CORBA annotations are not required.

6.4 Diagnosis and Repair

The diagnosis task in our implementation is based on the model-based diagnosis (MBD) paradigm [Rei87, dKW87] we have outlined in previous sections. In particular we use Reiter's hitting set algorithm [Rei87, GSW89] together with a propositional Horn clause theorem prover [Min88]. In order to minimize diagnosis time we only search for minimal cardinality diagnoses which can be easily obtained when using Reiter's algorithm. We only construct the hitting set graph until a level where the first diagnosis is computed. In most practical cases single fault diagnoses can be found. An upper bound for computing single fault diagnosis is determined by the amount of time required for checking consistency. In our case, we have logical rules that can be easily transformed to a set of horn clauses. Hence, time required for checking consistency is of the same order as the number of literals. Because we have to check all single fault diagnoses in the worst case and the size of the model, the worst case diagnosis time is bound by $O(|CO|^2 \cdot |C|^2)$. This bound comes from the use of an propositional horn clauses theorem prover.

After diagnosis those components that are responsible for a detected failure have to be killed and restarted. We have to take care of the fact that restarting one component might require restarting another component. This can be done by using the available information about strong dependencies between components. The components that have a strong dependency relationship with each other have to be restarted.

Hence, the steps for repair would be:

1. Compute the diagnoses.
2. Compute a set of components that have to be restarted. In this step we compute all components that strongly depend on components of a diagnosis.
3. Maximize the chance of repair by using a larger set of components to be restarted.

The following algorithm implements this behavior and has to be called by the $monitoring$ algorithm in step 4(c).

repair($CO, C, out, in, WDC, SDC), M, OBS$
Input: The SAM, its model M and observations OBS.

1. Compute diagnoses $D = diag(M, CO, OBS)$ where $D \subseteq CO$.
2. For each diagnosis $\Delta \in D$ compute the set of strongly dependent components, i.e., $R(\Delta) = \Delta \cup \{x | x \in CO : (x, \Delta) \in SDC\}$. Let $R = \{R(\Delta) | \Delta \in D\}$.
3. Reduce the set R by eliminating all elements that are subset of another set in R.
4. Select an element x from R.
5. Kill all processes that correspond to software components in x. Afterwards restart those processes.

We assume that faults leading to the different diagnoses Δ are independent. Whether $repair$ was successful or not in one point in time is detected by the $monitoring$ algorithm at a later point in time. Hence, in principle it is possible that $repair$ always tries the same correcting actions without resulting in a correct system state. This problem can be solved either by selecting the components to be restarted (step 4. of $repair$) non-deterministically or by storing informations about former actions. The latter solution avoids to take the same actions twice.

6.5 Experimental Results

The proposed diagnosis system has been implemented and tested on our mobile robot system. The robot control system runs on an embedded Pentium III PC with 850 MHz clock rate and 256 MB of RAM. The operating system on the PC is an ordinary Linux system.
The robot control system comprises several software modules. Each module runs as an independent process and implements different services. The services are based on CORBA. The

communication between these services are implemented either by direct CORBA method calls or by an event channel. The diagnosis system itself is implemented as a separate process to minimize the interference with the existing control system. The diagnosis system implements the four types of observers described in Section 6.3:

- *Periodic event production*: This observer looks for the regular appearance of specific events on the event channel.
- *Conditional event production*: This observer looks for the appearance of a specific event on the event channel after its trigger event was perceived.
- *Spawn processes*: This observer checks the number of processes spawned by a software module. This information is extracted from the *proc* file system of the OS.
- Periodic method calls: This observer looks for regular invocation of specific CORBA-methods.

The use of CORBA and OS services allows monitoring the robot control system without any impact to it. The model of the robot control system (software components, dependencies, observers) is specified in one XML file. Therefore, changes in the model or adaptation to other software systems are simple and straight forward. The diagnosis system is divided into three modules: a monitoring module (1), a diagnosis kernel (2) and a repair module (3).

The supervision of the robot control system and the interaction with the diagnosis system work in the following way. The monitoring module starts all necessary observers according to the model description and regularly checks for violations of the observers. If such a violation is detected, the diagnosis kernel is informed. The diagnosis kernel derives a diagnosis based on the model of the control system and the violated observations. The diagnosis is a set of malfunctioning software components which explain the improper behavior of the system. The derivation of a diagnosis is started after a certain amount of time, i.e., five seconds, within no more changes in the state of the observer are detected. This is done for stability reasons as it takes a certain amount of time for all observers to recognize an improper behavior. The diagnosis will be communicated to the repair module. It executes the appropriate repair action to recover the control system. Regarding to the set of malfunctioning components and their connections among them and to other components the repair module starts an appropriate repair action. The repair module first stops the malfunctioning modules and all modules which are strongly coupled to stopped modules. After that it restarts all stopped modules according to their modeled dependencies. This means to start that modules first other modules depending on. During the repair action no new diagnoses are derived. We do this for stability reasons as the repair action temporally may violate observers. After the repair action is completed the observers and the

diagnosis kernel are started again. After this stages the control system is again in the desired state.

For the evaluation of the proposed diagnosis system and its implementation, we did several experiments on our mobile robot. We introduced artificial faults into the robot control system and analyzed if the diagnosis system detected and located the fault and recovered the control system. We used two different fault scenarios:

- **Killing a component**: A certain software component is explicitly killed. This is equivalent to a crash of a certain component.
- **Deadlock a component**: A deadlock is introduced to a certain software component. This is equivalent to a malfunctioning software component.

Figure 6.5 shows the timing diagram for the diagnosis and repair of an introduced deadlock in the motion service (MO). After introducing the deadlock in MO the Periodic Event Observer for the event *MotionDelta* perceives that no more events are produced. After the waiting time the diagnosis kernel derived that MO is malfunctioning, denoted as *AB(MO)*. Instantly the repair process starts. The repair action comprises a stop of the Behavior Engine (BE), a stop of MO, and a restart of MO and BE. The restart of BE is necessary because BE is strongly coupled with MO. Again, after the waiting time, the diagnosis kernel derives the diagnosis that all components work properly now. Please note that no other components were affected by the repair process. The figure also shows the fact that suspending the diagnosis kernel during the repair is necessary as observers report additional improper observations, e.g., process observer. The relatively long time for the recovery is explained by the fact that stopping and starting of services could take a while because of the required starting, stopping and re-configuration of hardware components. The time for computing diagnosis is negligible because it has been less than 10 ms.

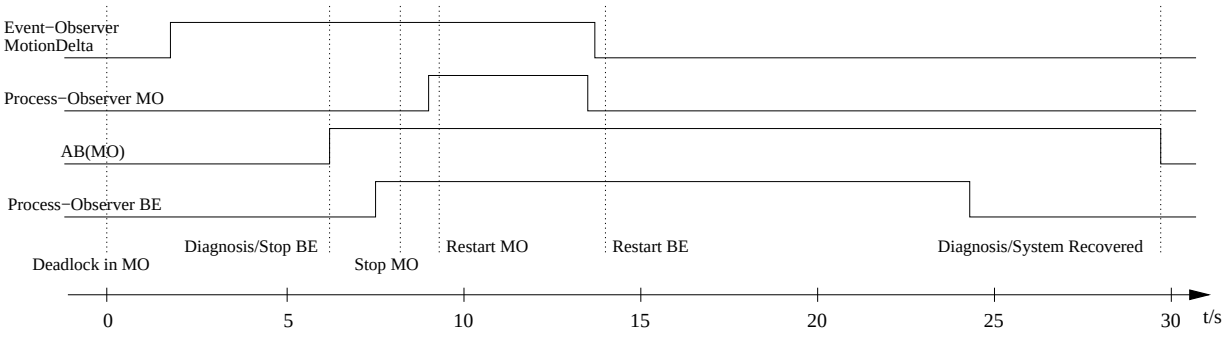

Figure 6.5: Timing diagram for diagnosis and repair of a deadlock in the motion service.

Figure 6.6 shows a more complex scenario. Here we introduced a deadlock in the CAN-service. After introducing the deadlock, MO and the sonar service SO produce no more data because they get no more data from CAN. This fact is perceived by the appropriate observers. Because of the model of the observations, the components and its connections the diagnosis kernel recognizes the malfunctioning CAN. The repair action is similar to the example above except that more components are involved. After repair, the control system is again in the desired state.

We conducted also two experiments where we killed a software component. In the first experiment we killed the laser service (LASER). The diagnosis system successfully detected the fault and recovered the control system by restarting BE, goal locator (GL) and LASER. The recovery took 68 s. In a second experiment we killed the world model (WM). The diagnosis system successfully detected and repaired the fault. During this experiment it was important that the whole process took only 20 s because the system located the fault in the WM and no other component was affected.
The affect of the diagnosis system to the runtime performance of the robot control system is negligible. The diagnosis system uses less than 1 % of the CPU time and less than 5 % of the memory.

6.6 Related Research

There are many proposed and implemented systems for fault detection and repair in autonomous systems. The Livingstone architecture by Williams and colleagues [MNPW98] was used on the space probe *Deep Space One* to detect failures in the probe's hardware and to recover from them. The fault detection and recovery is based on model-based reasoning. Model-based reasoning uses an abstracted logic-based formulation of the system model and the observations. The advantage is that well understood reasoning algorithms could be used. Model-based diagnosis also has been successfully applied for fault detection and localization in digital circuits and car electronics and for software debugging of VHDL programs [FSW99]. Dearden and colleagues [DC02] and Verma and colleagues [VGST04] used particle filter techniques to estimate the state of the robot and its environment. These estimations together with a model of the robot were used to detect faults. The advantage of this approach is that it accounts uncertainties of the robot's sensing and acting and in its environment because the most probable state is derived from unreliable measurements. A drawback of such methods is that one have to build a behavioral model of the system for each different fault one likes to detect. Rule-based approaches were proposed by Murphy and Hershberger [MH96] to detect failures in sensing and to recover from them. Additional sensor information were used to generate and test hypotheses to explain symptoms resulting from sensing failures. Roumeliotis et. al. [RSB98] used a bank of Kalman filter to

track specific failure models and the nominal model. The filter residuals were post-processed to produce a probabilistic interpretation of the operation of the system. Such methods are popular for linear systems affected by Gaussian noise. In [Gro04] a model-based approach for monitoring of component-based software was presented. The behavior of software components were modeled by Petri nets. Places in the net represent the state of a component. Transitions model the interactions with other components. These interactions, sending and receiving of messages, were used to locate a misbehavior in a software component. Liu and Coghill [LC04] used a qualitative representation to model the trajectory of a robot arm. Reasoning about these qualitative trajectories were used to detect and isolate faults of the robot arm. In [HW05] the authors used hybrid automata in combination with multi-hypotheses tracking to detect and locate faults in a miniature chemical plant.

6.7 Discussion

Previous research has dealt either with hardware diagnosis or diagnosis of software as part of the software engineering cycle. However, diagnosis of software and repair at runtime has never been an issue.

In this chapter we described a model-based diagnosis approach for detecting, locating and repairing software at runtime. For this purpose a modeling technique for representing software architectures which include components, control and data flow, and dependencies between components has been introduced. Moreover, the concept of observers, i.e., software which monitors the activity of the control software, together with their connections to the architecture models have been described in the paper. Finally, the chapter presented a repair algorithm and first empirical results of our implementation. These results show that software failures, e.g., deadlocks, can be detected and corrected at runtime.

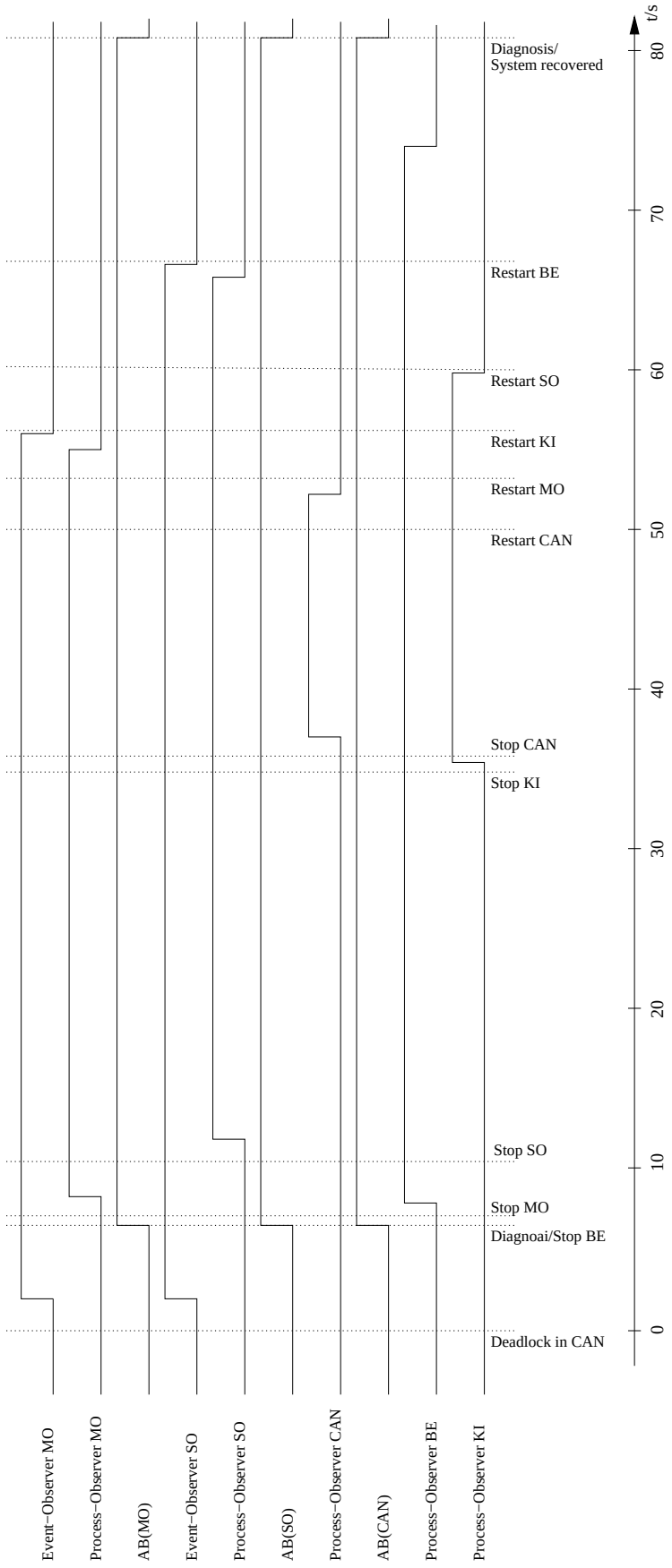

Figure 6.6: Timing diagram for diagnosis and repair of a deadlock in the CAN service.

Chapter 7

Model-Based Diagnosis for Hard- and Software

7.1 Introduction

An autonomous mobile robot comprises of great bunch of hardware and software components. The high number of different components and the heavily interaction between them cause a fairly high complexity of the system. Furthermore, because of complexity the probability of the occurrence of faults in the system during runtime increases. These facts are in conflict with the requirements of an autonomous mobile robots like robustness, long-term stability, and the capabilities of fulfilling a given task autonomously within an unknown environment. Even if the software and hardware of a truly autonomous mobile robot is well developed and tested faults and malfunctioning components can never be totally avoided. Such faults become even worse if there is almost no possibility for supporting actions by humans within a short period of time because of different reasons like distance or communication problems. Examples are a planetary rover on the Mars or an inspection robot in a nuclear power plant.

Hence, it is desirable that a robot is able to detect and repair faults of its hardware and software autonomously. This task requires the robot to reason about its underlying system in order to identify a misbehavior, locate the root cause for the misbehavior and to recover autonomously from the root cause, i.e., the faults. Recovery from faults include a simple restart of components, a reconfiguration of the components or a controlled degradation of the robots functionality. The challenge herein is that all the detection, localization and recovery have to be performed at runtime while the robot actually perform its mission.

Therefore, a dedicated monitoring and diagnosis system is crucial to reach the above requirements. But there are additional requirements for a diagnosis system in an autonomous mobile robot. For example, the introduction of a diagnosis system should not cause heavy changes in

the hard and software of the target system. Furthermore, the diagnosis system should not affect the overall behavior of the robots. As computational resources are usually very limited in mobile robots the requirements for memory and computation for the diagnosis system should be as low as possible.

In order to fulfill all the above requirements a model-based solution is preferable. Model-based diagnosis have been successfully applied to diagnosis of integrated circuits and the debugging of software. But the application mostly took place during the development process. Therefore, the idea of the application of a model-based approach at runtime is quite novel. Model-based diagnosis use a model of the correct behavior of a system and current observations on the system to detect misbehaviors and to locate the cause of the fault. It is a general paradigm. Therefore it is the ideal approach for the supervision of a system comprising of hardware and software.

The important step in the application of model-based diagnosis is the creation of an appropriate abstract model of the correct behavior of the system and the determination of useful facts about the system that could be observed by the robots sensors. The modeling of software components and their interactions are well understood. But if one like to diagnosis of a mixed system of hardware and software also some aspects of the physical system and its interaction with its environment have to be modeled in a qualitative manner.

In this chapter we will (1) introduce the application area robotics for model-based systems approach, (2) present first ideas of modeling the robot's sub-systems and its surrounding environment, (3) identify problems that occur when modeling the systems and trying to fulfill the previously discussed requirements, and (4) discuss the grand challenges of modeling and diagnosis in mobile and autonomous robotics. The chapter is organized as follows. First, we introduce a running example from robotics. We present a model that has the capabilities of identifying faults on an abstract level and discuss open problems and challenges. We conclude the chapter by discussing related research and summarizing the content. Some of the ideas were previously presented in [SW05a].

7.2 Example — A Case From Robotics

In order to give a short introduction into the problem domain, we shortly introduce our mobile robot platform comprising hardware and software components. Figure 7.1 shows some parts of the whole system which closely interact with the environment. The components for sensor fusion, world modeling and high-level decision making are omitted. The robot uses an omni-directional drive which comprises four omni-wheels, and four motors and wheel encoders. The drive is controlled by a drive controller which receives the desired movement commands in the form of a motion vector and a rotation angle relative to the robot. Based on these commands the

controller controls the speed of the motors. Furthermore the controller provides regularly odometry data based on the measurements of the wheel encoders. The controller itself comprises hardware and software components. The drive controller gets its commands from the behavior engine which is responsible for the execution of actions like move to a position. The appropriate action is selected by high-level decision making based on classical planning. Furthermore, the robot is equipped with an omni-directional camera. Based on the provided images a movement estimator measures a movement of the robot by calculating a significant optical flow in the image. A laser scanner provides range information around the robot. This information is used to determine the positions of obstacles around the robot.

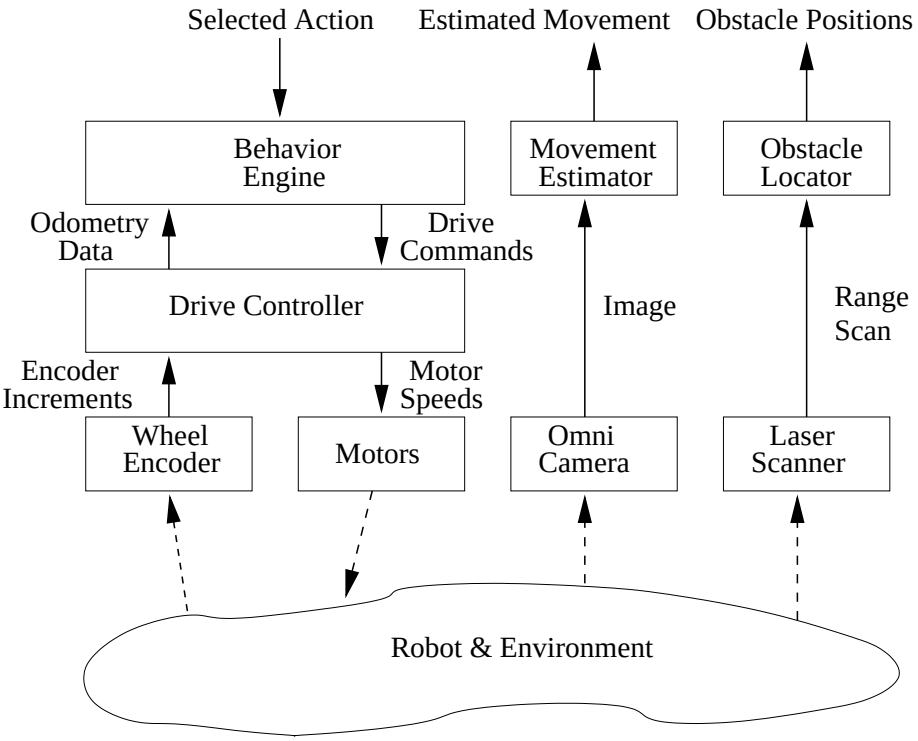

Figure 7.1: Interactions between components of a mobile robot.

Because of the current lifetime and the extensive use of our robots during the last years we observe sporadic faults in both the hardware and the software of the robots drive. We will give two examples for such faults to motivate how these faults can be detected, located and repaired by a model-based approach. Consider the following situation. At certain times the circuits for the wheel encoder hang up. Because of this, the drive controller provides odometry data which report no movement of the robot even if the robot moves in reality. If one uses a model of the correct interactions of the components in combination with current observations within the system the fault and its cause can be easily detected. For instance, if we know that the

system works properly, the behavior engine calculates a desired movement. The drive controller converts this desired movement into appropriate motor speeds. It also provides an estimation of the current movement of the robot by odometry data. Furthermore, if the robot moves, the camera image have to change and also the position of detected obstacles in the laser scans. Suppose, the odometry fails because of the above facts. The drive controller reports a zero movement which suggests that the robot does not move. But if the robots motors are working as expected the robot may move correctly. Therefore, this might be a wrong conclusion. But if we combine this observation with the observation from the camera and the laser range-finder, we conclude that the odometry is malfunctioning because the laser range-finder and the camera report the desired movement. This reasoning process is quite similar to the way a human would detect and locate a fault. Also a appropriate repair action can be derived. A simple reset of the encoder circuit removes the fault. In Section 6 we presented a model-based modeling paradigm for the control software of mobile robots. The above example is an extension towards the diagnosis of robots hardware and shows that more work have to be done in modeling of the environments and its interaction with the robot. In the next section we introduce the model, the observation and the reasoning process in more details.

7.3 Modeling for Diagnosis

The model we introduce in this section is for being used in consistency-based diagnosis [Rei87] and describes the underlying behavior of the robot in an very abstract way. There are several different approaches for modeling highly dynamical systems for diagnosis including finite state machines or probabilistic automatons. Most of them use a discrete time representation like automatons and state machines. For our model of the hardware we use an abstraction of the real situation which avoids dealing with time. Such abstract models have been successfully used in other areas, e.g., representing knowledge in the automotive domain [MSS95, MS96].

For modeling we choose the component-oriented modeling paradigm. Hence, we represent the behavior of components and their interactions. Figure 7.2 shows an overview on the modeling of the running example.

Motor A motor M is said to be working if it receives a driving command. A working motor causes the axis and as a consequence the wheels to rotate which moves the robot from one place to another. However, this cause-effect chain cannot be inverted. For example, the robot can move without the driving command because of external influences like a collision where one robot is moving the other. Or one robot tries to go ahead while being stuck to an obstacle. In this case the wheels are rotating but the robot is not moving. This fact is caused by the slippage of the wheels.

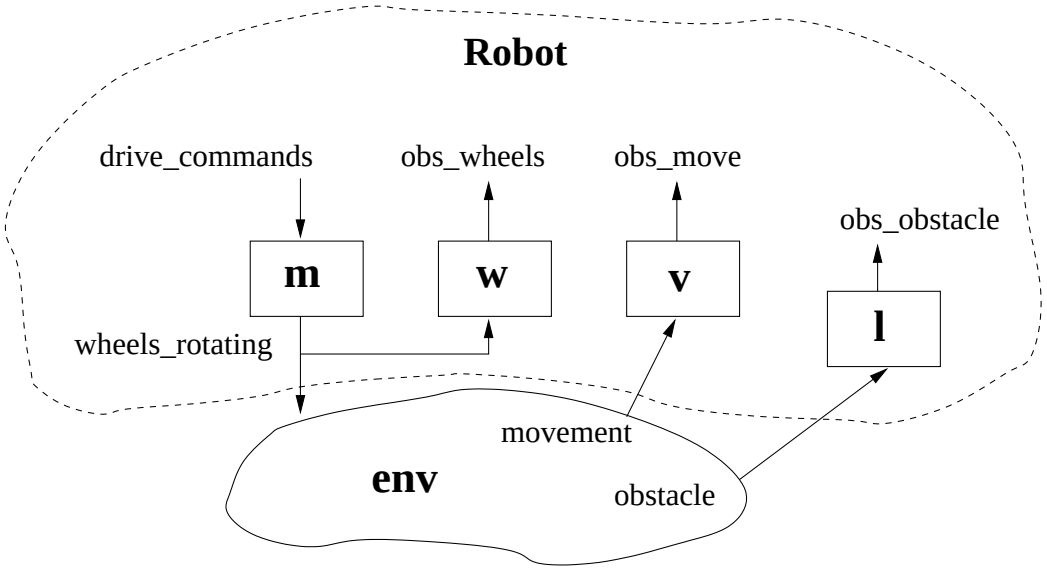

Figure 7.2: Observations and model for the running example.

We represent the discussed cause-effect relationship using two rules. The first one expresses that a drive command is necessary for switching the motors on (and vice versa).

$$\neg Ab(M) \rightarrow (drive\_command(M) \leftrightarrow motor\_on(M))$$

The second rule is for stating that a switched-on motor is one of the causes for rotating wheels.

$$motor\_on(M) \rightarrow wheels\_rotating(M)$$

Wheel encoder A wheel encoder W observes the behavior of its corresponding wheel. Whenever the wheel is rotating, the rotation speed is given back as an observation. Unfortunately a rotating wheel and thus a wheel encoder giving back a speed does not imply that the robot is moving. For example, consider the situation where a robot is stuck to an obstacle. However, every time the wheel encoder detects a rotation the wheels are really rotating.

$$\neg Ab(W) \rightarrow (wheels\_rotating(W) \leftrightarrow obs\_wheels(W))$$

Vision If vision V perceives a movement via its camera, it generates a corresponding observation. This is expressed by an input proposition where the environment is setting the truth value.

$$\neg Ab(V) \rightarrow (movement \leftrightarrow obs\_move(V))$$

After stating the components' behavior, we have to formalize the structure of the system. For this purpose, we assume a motor m, a wheel encoder w, and a vision system v within the corresponding rules.

$$\begin{aligned}
&\neg Ab(m) \rightarrow (drive_command(m) \leftrightarrow motor_on(m)) \\
&motor_on(m) \rightarrow wheels_rotating(m) \\
&\neg Ab(w) \rightarrow (wheels_rotating(w) \leftrightarrow obs_wheels(w)) \\
&\neg Ab(v) \rightarrow (movement \leftrightarrow obs_move(v))
\end{aligned}$$

Because of the fact that the wheels and the wheel encoder are both mounted on the axis of the robot which is connected with the motor, the wheel rotation for the motor m and the encoder w is the same which is expressed by the rule:

$$wheels\_rotating(m) \leftrightarrow wheels\_rotating(w)$$

The model for the connection of the vision system with the motor unit is not that easy because there is no direct connection between them which belong to the robot. Instead we are only able to argue about a chain of cause-effect relationships like follows. The motor is driving the wheels which move the robot within its environment. Because of changes within the perceived image, the vision system detects the movement. This relationship can only be established whenever the environment does not interact in an undesired or unexpected way. For example, as stated previously, an obstacle can prevent the robot of moving but the wheels are rotating. Or there is a movement that is caused by a person which carries the robot from one place to another. In both cases the observations regarding wheel movements are contradicting the visually perceived observations of the robot. To overcome the described problem it is necessary to model the environment explicitly.

The relevant parts of the model of the environment env can be expressed by rules which link the robot's actuator propositions to the propositions which correspond to sensors. For our example we have to provide a link from the rotating wheels to the proposition $movement$ which is used by the vision system as an input. We can do this by formulating that rotating wheels imply a movement providing that they are no other external influences like carrying the robot or obstacles preventing the robot to move.

$$\neg Ab(env) \rightarrow (wheels\_rotating(m) \rightarrow movement)$$

For each actuator a similar rule have to be provided. These rules allow to robot to reason not

only about its components but also on states of the environment which influences the functionality of the robot.

To illustrate the capabilities of our model, we introduce the observations *OBS* which describe a situation where the wheels are rotating but the motion is blocked by an obstacle.

$$OBS = \left\{ \begin{array}{l} drive\_command(m), \\ obs\_wheels(w), \\ \neg obs\_move(v) \end{array} \right\}$$

If using the model, i.e., the system description *SD*, and the observations *OBS* together with the set of components *COMP* $= \{env, m, w, v\}$, we can derive the set of conflicts and diagnosis. We use Reiter's diagnosis theory [Rei87] for this purpose and obtain two conflicts $\{m, env, v\}, \{w, env, v\}$ from which we derive 1 double fault diagnosis $\{m, w\}$ and 2 single fault diagnoses $\{env\}, \{v\}$. In order to further distinguish the two single fault diagnoses it is necessary to add new observations. This can be done by introducing a new sensor, e.g., a laser range-finder that gives back information about the occurrence of obstacles within a certain area.

The model for a laser range-finder has to assign the observation about obstacles to the occurrence of obstacle in an environment.

$$\neg Ab(L) \rightarrow (obstacle \leftrightarrow obs\_obstacle)$$

Moreover, we have to extend the model for the environment. It has to be stated that whenever the robot can move, an obstacle is not in reach.

$$\neg Ab(env) \rightarrow (movement \rightarrow \neg obstacle)$$

We now extend our system by introducing a new component laser range-finder l, i.e., *COMP'* $= \{l\} \cup$ *COMP*, and add a new observation to the set of observations, i.e., *OBS'* $= \{obs\_obstacle\} \cup$ *OBS*. From the new model SD' which comprises all rules from *SD* and the new ones, *OBS'* and *COMP'* we obtain additional conflicts $\{l, env, v\}, \{m, env, l\}, \{w, env, l\}$. Hence, we derive only one single fault diagnosis $\{env\}$. If preferring smaller diagnoses, a robot using the described knowledge base would conclude that its systems are working correctly and the problem is caused by external influences. Note that the conclusion can also be used to diagnose the robot's software. For example, if the control software should avoid collisions with obstacles, getting stuck because of an obstacle should never happen. Therefore, the control software is not working correctly, and further steps have to be taken into account.

7.4 Problems and Challenges

Although, fault detection and localization can be done in a straight forward way, this is not the case when dealing with autonomous recovering from failure. Models for fault localization of hardware and software has been described in literature but there are not many papers which deal with repair and re-configuration which is necessary to recover. The following list identifies problem and challenges that have to be tackled in order to fulfill the requirements of autonomous systems.

Coupling different models When dealing with real-world systems we have to provide models both for hardware and software. Although, the amount of software of devices and systems is increasing, there are always hardware components involved which directly interact with the environment. Hence, it is required to couple different models at different level of abstraction in order to identify root causes for a detected misbehavior. In principle model-based diagnosis is well adapted for this purpose because it is always possible to integrate models at least by using conflicts or using the outcome of one model as an input for another. An example for the latter coupling is the following: Consider the last example of Section 7.3 where we argued that the diagnosis result, i.e., an obstacle caused the misbehavior, can be used as an input for diagnosing the robot's control software which should avoid such situations. Hence, investigating possibilities and theories for coupling models is a challenge for future research.

Repair Although, there is literature available which deals with repair and the selection of repair actions, the problem can be considered as an still open problem. In most cases, repair is done quite simple by assigning corresponding repair actions to components. However, this is not enough in general especially when dealing with an autonomous system which has to survive in an unknown environment. The reason is that someone has to provide all possible repair actions in advance which can hardly be done for complex systems interacting with other systems in a highly complex environment. In addition, the requirement of truly autonomous and mobile systems does not allow to simply replace a component by another. In our running example the system has to be aware about resets for solving the problem of the wheel encoder circuits. Another example is the following: Consider a situation where one motor of the omni-wheel drive is broken. In principle it is possible to steer the robot by slightly modifying the control software of the drive controller. Hence, the robot should perform the appropriate actions because maintenance activities like replacing the broken motor is might be not possible.

The challenge here is to provide a theory and model guidelines that allow for coming up with repair actions for certain diagnoses which are not explicitly known in advance. Such repair actions can be derived from the representation of functionality of systems and their parts. Hence,

solutions of configuration and re-configuration problems are might be promising starting points.

Meta-modeling Because of the previous problems and challenges it might be necessary to provide the same reasoning capabilities on the models. For example, when changing the behavior of the system because of recovery actions like changing the behavior of the omni-drive to a differential drive, it is required to change the diagnosis model. Hence, reasoning capabilities which allow for modifying a model are required. As far as we know, there is no work within the model-based systems community which deals with meta-modeling or model generation from generic models. Moreover, it might be necessary for an autonomous robot to establish meta reasoning capabilities whenever required and also on previously generated meta-models. The challenge is to provide first steps towards meta representation and reasoning for model-based systems.

Gaining knowledge from observation Another challenge which is highly demanding would be to generate models or at least to gain some knowledge from previously obtained observations. A mobile and autonomous system might require to adapt its behavior to the environment. Hence, changes of the behavior which causes changes of the underlying models would be necessary. A system that can generate models automatically from observations which include observations regarding the environment, interactions, and internal states would be truly autonomous. However, a first step would be to investigate model adaption that goes beyond adapting fault probabilities because of previously gained experiences.

7.5 Discussion

Previous research has dealt either with hardware diagnosis or diagnosis of software as part of the software engineering cycle. However, diagnosis of hard and software and repair at runtime have never been an issue. The related research discussed in the previous section is also relevant to this section. This section described a model-based diagnosis approach for detecting, and locating faults of a mobile robot platform at runtime which includes partially reasoning about the current state of the environment. Moreover, we identify some open problems and challenges that have to be tackled in order to provide a really autonomous diagnosis and repair system that can respond even to faults that were not been considered when developing the system. Two main challenges can be identified. One is the coupling of different models which capture either different aspects of the system or the same aspects but using a different level of abstraction. The other challenge is to provide representation and reasoning techniques that allow for generating repair actions which are based on the current diagnoses and the current state of the robot. These repair actions cannot be tied to a specific diagnosis or a component but has to be autonomously generated from the

provided functionality of the robot and its desired functionality. Moreover, the repair actions might change the diagnosis model.

Chapter 8

Shortcomings and Future Research

The proposed framework for intelligent control of autonomous mobile robots has shown its capability to robustly control a robot in various tasks and environments. Furthermore, it has been successfully evaluated in two real-world applications, the RoboCup Middle-Size League and service robotics. So far we have answered most of the questions we raised in the introduction and have provided solution to the related problems.

Obviously, we are not able to answer all the questions emerging from the control of robots nor it is able to provide a perfect solution to all the related problems. In the remainder of this section we will discuss some of the shortcomings and the insufficient solutions for some of the problems. The reminder of this section is similar organized as the problem statement in the introduction.

- **Flexibility and reuse of components:** The used basic Miro-framework and our extensions have been proofed as a flexible base for research in robotics. Furthermore, we successfully have used the framework on two different robot platforms. One shortcoming of the framework is the extensive use of CORBA which slows down the application in particular situation. The communication mechanisms, e.g., event channel, used in our framework have to be systematically analyzed and redesigned under the guidelines of software engineering in order to improve the performance of the system.

  Furthermore, more templates of general usable algorithms have to be integrated in the framework. Despite most of the basic components are shared among the different research groups, most of the components for sensor-fusion, localization and planning are re-implemented by each group. There is a clear demand for share-able basic algorithms. Such a attempt requires a good knowledge of the different requirements and the definition of appropriate interfaces.

  Finally, appropriate interfaces or implementations for a large set of new sensors have to be added to the framework in order to be able to use “state of the art” equipment.

- **Planning and reasoning for complex tasks:**

  The value of a deliberative component in the control system of a robot is visible. The STRIPS-based planning system of the framework is capable to derive plans for almost every task we have evaluated so far.

  But in the future more recent planning algorithms have to be integrated in the framework in order to improve the handling of uncertainty in the deliberative layer and to improve the planning performance for tasks with a high complexity, e.g. a great number of possible actions, positions and items. Also multi-agent planning is a issue for the future. This topic will discussed below.

- **General, expressive and intuitive task description:**

  So far we are only able to specify tasks for a single robot. Cooperation and communication between multiple robots are rudimentary part of the lower levels of the framework but such actions are not supported by statements in the task description. In the future, it will be desirable to specify a task for a group of robots which carry out a task in cooperation. Such an extension of the task description inherently demands for the integration of recent multi-agent planning capabilities into the framework. Moreover, statements for communication have to be part of the task description.

  Furthermore, the current task description is mainly based on the STRIPS representation. Although, it incorporates some extensions like quantifiers it still lacks of expressiveness for some tasks. Temporal relations and some kind of support of uncertain knowledge are possible extensions one may think about.

  Finally, an exchange format for the knowledge of a robot and the description of the capabilities of a robot among heterogeneous robots will further improve the quality of the task description. This is true especially for tasks were robots have to cooperate in order to fulfill a task.

- **Robust task execution in noisy and dynamic environments:**

  Plan invariants have been shown as a appropriate method to encode knowledge of the structure of a task. Up to now the invariants are encoded by hand. We presented an algorithm which is able to generate conditions similar to the presented invariants. But these algorithm lacks of general applicability.

  Further research should be done on the question if such invariants automatically can be derived from a task description and how such a derivation can be done. Furthermore, we believe that more encoded knowledge about the structure of the tasks and the world

can improve the quality of such invariants and could ease the automated generation of invariants.

Finally, it is an interesting question if it is possible to extract qualitative knowledge form raw sensor data without taking the way via a quantitative noise-sensitive representation.

- **Fault-tolerance:**

  The diagnosis system which we have integrated into the framework is able to detect, localize and repair faults in the control software of the robots. But the used component models have to be extended in order to increase the number and quality of the detect faults.

  Furthermore, the diagnosis system have to be extended in the direction of diagnosis of the robot's hardware. For this purpose models of the interaction of the robot with the physical world are needed. Furthermore, most of this models will model the behavior of sensors and actors which operate in the continuous world and hardly can be modeled on an abstract level. Therefore, continuous models and techniques for diagnosis of continuous systems have to be integrated into the diagnosis system.

  So far the repair process for a detected fault in the software comprises the restart of the faulty and all its strongly related components. Which is an appropriate action if the control software comprises of several independent applications like in our framework? But if the control software comprises only a single application the repair action is equivalent to a complete restart of the software. In the future work should be done on a rollback mechanism which rolls back the execution to a earlier point in time where the fault has its root cause. Then the root cause of the fault can be eliminated and the correct execution can be resumed. A interesting question is how to realize such a approach on a system which interact with its environment physically in real-time.

  Finally, work should be done on the description of the structure of and relations within the hardware and the software systems of the robot. If such relations and the behavior of the components of the systems are modeled more sophisticated repair actions can be derived. Such repair action roughly can be separated into two groups. The repair actions of the first category are derived by a planning process and denotes a sequence of actions which drives the system back to a nominal state. Such repair action are appropriate for transient faults like segmentation faults. The members of the other category are derived by a reconfiguration process. If some level of redundancy is part of the system such repair action can handle permanent faults like broken motors.

Chapter 9

Summary and Conclusion

In this work we presented a framework which enables the robust intelligent control of autonomous mobile robots for various complex tasks in dynamic real-world environments.

The demand for such a framework was motivated by the increasing number of task in industry and everyday live which are carried out by autonomous mobile robots in regular manner. Furthermore, today mobile robots serve as a real-world testbed for various research areas like computer science, economy and biology. These facts make autonomous mobile robots interesting for robotics and AI researcher and post a wide range of new scientific questions.

During the work for this thesis we incrementally developed the proposed control framework in order to answer the raised question and to provide solutions to the problems which arise from the deployment of autonomous robots in the real world.

We discussed two example application areas for autonomous mobile robots, the RoboCup Middle Size League and service robotics, in which we successfully deployed and evaluated the proposed framework.

The overall question we have tried to answer in this thesis was, how can we robustly and flexibly control a robot for different tasks in dynamic environments under the presence of noise, uncertainty and faults. The problems we have to solve in order to answer this overall question lead to five major contributions:

- We developed a modular flexible software and hardware framework which allows us to carry out various research in the domain of mobile robots. The developed framework is able to serve as a basis for various tasks and was successfully deployed in the RoboCup and the service domain. This framework is a strong foundation for our past and ongoing research.

- We have motivated that a strong deliberative component have to be part of a control framework of a robot in order to enable a robot to carry out complex tasks. We discussed how

the above framework has been enriched with reasoning and planning capabilities which allows flexible deliberative control.

- We have shown the benefit of a flexible logic-based task description. We have integrated the task description into the framework is appropriate for various tasks, expressive enough also for complex tasks and finally intuitively readable for humans.

- We have shown the problems which arise if a robot is deployed in a noisy, uncertain and dynamic environment. We have enriched the control framework with a robust plan execution which is able quickly to react to unforeseen situation. Furthermore, we have shown how robustly to bridge the gap between the qualitative and the quantitative knowledge representation for the robot in the case of a noisy and uncertain environment.

- We discussed the issue of faults at runtime in the software and hardware of mobile robots. We presented a diagnosis system which is capable to detect, localize and repair some faults in the control software at runtime. The diagnosis system was integrated into the framework in order to improve its fault-tolerance. Moreover, we present results of the diagnosis system from experiments carried out in the RoboCup environment. Finally, we discussed how the diagnosis system can be extended in order to be able to handle faults in the robot hardware.

All this features were integrated into a framework for the robust intelligent control of autonomous mobile robots. The complete framework has been successfully deployed in the RoboCup and service robotic scenario. The framework enabled our robots to carry out different tasks, e.g. robot soccer or book deliveries, in general environments under the presence of different disturbing issues like noise and faults.

Finally, we discussed some of the shortcomings of the so far implemented framework and point out some directions for future research.

Bibliography

[BAB+01] Michael Beetz, Tom Arbuckle, Thorsten Belker, Maren Bennewitz, Wolfram Burgard, Armin B. Cremers, Dieter Fox, Henrik Grosskreutz, Dirk Haehnel, and Dirk Schulz. Integrated plan-based control of autonomous service robots in human environments. *IEEE Intelligent Systems*, 16(5):56–65, 2001.

[Bea00] Mark F. Bear. *Neuroscience: Exploring the brain*. Williams and Wilkins, Baltimore, MA, 2000.

[Bee02] Michael Beetz. *Plan-Based Control of Robotics Agents, Improving the Capabilities of Autonomous Robots*, volume 2554 of *Lecture Notes in Artificial Intelligence*. Springer, 2002.

[BF95] Avrim Blum and Merrick Furst. Fast planning through planning graph analysis. In *Proceedings of the 14th International Joint Conference on Artificial Intelligence (IJCAI 95)*, pages 1636–1642, 1995.

[Bis95] Christopher M. Bishop. *Neural Networks for Pattern Recognition*. Oxford, UK: Oxford University Press, 1995.

[BJNT06] Asgard Bredenfeld, Adam Jacoff, Itsuki Noda, and Yasutake Takahashi, editors. *RoboCup 2005: Robot Soccer World Cup IX*, volume 4020 of *Lecture Notes in Artificial Intelligence*. Springer, 2006.

[BKLS05] Harald Burgsteiner, Mark Kröll, Alexander Leopold, and Gerald Steinbauer. Movement prediction from real-world images using a liquid state machine. In *18th International Conference on Industrial and Engineering Applications of Artificial Intelligence and Expert Systems.*, volume 3533 of *Lecture Notes in Artificial Intelligence*, pages 121–130, Bari, Italy, 2005. Springer.

[BMM98] Marco Baioletti, Stefano Marcugini, and Alfredo Milani. Encoding planning constraints into partial order planning domains. In Anthony G. Cohn, Lenhart Schubert,

and Stuart C. Shapiro, editors, *KR'98: Principles of Knowledge Representation and Reasoning*, pages 608–616. Morgan Kaufmann, San Francisco, California, 1998.

[Bro86] Rodney. A. Brooks. A robust layered control system for a mobile robot. *IEEE Jornal of Robotics and Automation*, RA-2(1):14–23, 1986.

[Bru01] Herman Bruyninckx. Open robot control software: the OROCOS project. In *IEEE International Conference on Robotics and Automation (ICRA)*, volume 3, pages 2523–2528, 2001.

[BSK03] Herman Bruyninckx, Peter Soetens, and Bob Koninckx. The real-time motion control core of the Orocos project. In *IEEE International Conference on Robotics and Automation (ICRA)*, volume 2, pages 2766–2771, 2003.

[CGI+02] Claudio Castelpietra, Alice Guidotti, Luca Iocchi, Daniele Nardi, and Riccardo Rosati. Design and Implemantation of Cognitive Soccer Robots. In *RoboCup 2001: Robot Soccer World Cup V*, volume 2377 of *Lecture Notes in Computer Science*. Springer, 2002.

[DC02] Richard Dearden and Dan Clancy. Particle filters for real-time fault detection in planetary rovers. In *Proceedings of the Thirteenth International Workshop on Principles of Diagnosis*, pages 1 – 6, 2002.

[DFL02] Frank Dylla, Alexander Ferrein, and Gerhard Lakemeyer. Acting and deliberating using golog in robotic soccer – A hybrid approach. In *Proc. 3rd International Cognitive Robotics Workshop (CogRob 2002)*. AAAI Press, 2002.

[DFL03] Frank Dylla, Alexander Ferrein, and Gerhard Lakemeyer. Acting and Deliberating using Golog in Robotic Soccer - A Hybrid Architecture. In *The Third International Workshop on Cognitive Robotics*. AAAI, 2003.

[DGN01] Markus Dietl, Jens-Steffen Gutmann, and Bernhard Nebel. Cooperative sensing in dynamic environments. In *Proceedings of the IEEE/RSJ International Conference on Intelligent Robots and Systems (IROS'01), Maui, Hawaii*, 2001.

[Dij76] Edsger W. Dijkstra. *A Discipline of Programming*. Series in Automatic Computation. Prentice-Hall, 1976.

[dKW87] Johan de Kleer and Brian C. Williams. Diagnosing multiple faults. *Artificial Intelligence*, 32(1):97–130, 1987.

[Elm90] Jeffrey L. Elman. Finding structure in time. *Cognitive Science*, 14:179–211, 1990.

[Ets01] Konrad Etschberger. *Controller Area Network (CAN) Basics, Protocols, Chips, Applications*. IXXAT Automation, 2001.

[FBDT99] Dieter Fox, Wolfram Burgard, Frank Dellaert, and Sebastian Thrun. Monte carlo localization: Efficient position estimation for mobile robots. In *AAAI/IAAI*, pages 343–349, 1999.

[FFL04] Alexander Ferrein, Christian Fritz, and Gerhard Lakemeyer. On-line decision-theoretic golog for unpredictable domains. In *Proc. of 4th International Cognitive Robotics Workshop*, 2004.

[FHN81] Richard E. Fikes, Peter Hart, and Nils Nilsson. Learning and Executing Generalized Robot Plans. In Bonnie L. Webber and Nils J. Nilsson, editors, *Readings in artificial intelligence*, San Francisco, CA, USA, 1981. Morgan Kaufmann Publishers Inc.

[FL98] Maria Fox and Derek Long. The automatic inference of state invariants in tim. *Journal of Articial Intelligence Research*, 9:367–421, 1998.

[FL00] Maria Fox and Derek Long. Utilizing automatically inferred invariants in graph construction and search. In *Artificial Intelligence Planning Systems*, pages 102–111, 2000.

[FL03] Maria Fox and Derek Long. *PDDL2.1: An Extension to PDDL for Expressing Temporal Planning Domains*. University of Durham, UK, 2003.

[FLBM01] Maria Fox, Derek Long, Steven Bradley, and James McKinna. Using model checking for pre-planning analysis. In *AAAI Spring Symposium Model-Based Validation of Intelligence*, pages 23–31. AAAI Press, 2001.

[FN71] Richard. E. Fikes and Nils J. Nilsson. STRIPS: A new approach to the application of theorem proving to problem solving. *Artificial Intelligence*, 2(3-4):189–208, 1971.

[Fra03] Gordon Fraser. AI-Planning System for Robotic Soccer. Master's thesis, Graz University of Technology, 2003.

[FSW99] Gerhard Friedrich, Markus Stumptner, and Franz Wotawa. Model-based diagnosis of hardware designs. *Artificial Intelligence*, 111(2):3–39, 1999.

[FSW04a] Gordon Fraser, Gerald Steinbauer, and Franz Wotawa. Application of qualitative reasoning to robotic soccer. In *Working Papers of the 18th International Workshop on Qualitative Reasoning (QR-04)*, 2004.

[FSW04b] Gordon Fraser, Gerald Steinbauer, and Franz Wotawa. A modular architecture for a multi-purpose mobile robot. In *Innovations in Applied Artificial Intelligence, IEA/AIE*, volume 3029 of *Lecture Notes in Artificial Intelligence*, pages 1007–1014, Ottawa, Canada, 2004. Springer.

[FSW05] Gordon Fraser, Gerald Steinbauer, and Franz Wotawa. Plan Execution in Dynamic Environments. In *Proceedings of the 18th Conference on Industrial and Engineering Applications of Artificial Intelligence and Expert Systems, IEA/AIE*, volume 3533 of *Lecture Notes in Artificial Intelligence*, pages 208–217, Bari, Italy, 2005. Springer.

[GNT04] Malik Ghallab, Dana Nau, and Paolo Traverso. *Automated Planning Theory and Proctice*. Morgan Kaufmann Publishers, 2004.

[Gro04] Irene Grosclaude. Model-based monitoring of component-based software systems. In *15th International Workshop on Priciples of Diagnosis*, pages 155–160, Carcassonne, France, 2004.

[GS98] Alfonso Gerevini and Lenhart K. Schubert. Inferring state constraints for domain-independent planning. In *AAAI/IAAI*, pages 905–912, 1998.

[GS00] Alfonso Gerevini and Lenhart K. Schubert. Discovering state constraints in DISCOPLAN: Some new results. In *AAAI/IAAI*, pages 761–767, 2000.

[GSW89] Russell Greiner, Barbara A. Smith, and Ralph W. Wilkerson. A correction to the algorithm in Reiter's theory of diagnosis. *Artificial Intelligence*, 41(1):79–88, 1989.

[GVS+01] Brian P. Gerkey, Richard T. Vaughan, Kasper Stroy, Andrew Howard, Gaurav S. Sukhatme, and Maja J Mataric. Most valuable player: A robot device server for distributed control. In *Proceedings of the IEEE/RSJ International Conference on Intelligent Robots and Systems (IROS 2001)*, pages 1226–1231, 2001.

[GWM00] Anirudh Gupta, Yun Wang, and Henry Markram. Organizing principles for a diversity of gabaergic interneurons and synapses in the neocortex. *Science*, 287:273–278, 2000.

[Hop82] John J. Hopfield. Neural networks and physical systems with emergent collective computational abilities. In *Proceedings of the National Academy of Science*, volume 79, pages 2554–2558, 1982.

[HW05] Michael Hofbaur and Franz Wotawa. A diagnosis-based causal analysis method for concurrent hybrid automata. In *Proceedings of the 16th International Workshop on Principles of Diagnosis (DX05)*, pages 81–87, 2005.

[Jae01] Herbert Jaeger. The echo state approach to analysing and training recurrent neural networks. Technical Report 148, GMD, 2001.

[JW99] Micheal I. Jordan and Daniel M. Wolpert. Computational motor control. In M. Gazzaniga, editor, *The Cognitive Neurosciences*. MIT Press, Cambridge, MA, 1999.

[KBM98] David Kortenkamp, R. Peter Bonasso, and Robin Murphy, editors. *Artificial Intelligence and Mobile Robots. Case Studies of Successful Robot Systems*. MIT Press, 1998.

[KC92] G. Kelleher and A. G. Cohn. Automatically synthesising domain constraints from operator descriptions. In Bernd Neumann, editor, *Proceedings of the 10th European Conference on Artificial Intelligence*, pages 653–655, Vienna, August 1992. John Wiley and Sons.

[KM98] Kurt Konolige and Karen Myers. *The Saphira architecture for autonomous mobile robots*, chapter 9. In Kortenkamp et al. [KBM98], 1998.

[KMU+04] Gerhard Kraetzschmar, Gerd Mayer, Hans Utz, Philipp Baer, Martin Claus, Ulrich Kaufmann, Markus Lauer, Simon Natterer, Sebastian Przewoznik, Roland Reichle, Christoph Sitter, Florian Sterk, and Günther Palm. The ulm sparrows 2004. In *International RoboCup Symposium*, 2004.

[Koh01] Teuvo Kohonen. *Self-Organizing Maps*. Springer-Verlag, 3 edition, 2001.

[Kon97] Kurt Konolige. Colbert: A language for reactive control in sapphira. In *Advances in Artificial Intelligence*, volume 1303 of *Lecture Notes in Computer Science*. Springer, 1997.

[Kon00] Kurt Konolige. A gradient method for realtime robot control. In *Proceedings of the IEEE/RSJ International Conference on Intelligent Robots and Systems (IROS)*, volume 1, pages 639 – 646, 2000.

[KS98] Henry A. Kautz and Bart Selman. The role of domain-specific knowledge in the planning as satisfiability framework. In *Artificial Intelligence Planning Systems*, pages 181–189, 1998.

[LBS96] Xiao-Rong Li and Yaakov Bar-Shalom. Multiple-model estimation with variable structure. In *IEEE Trans. Automatic Control*, volume 41, pages 478–494, 1996.

[LC04] Honghai Liu and Gorge M. Coghill. Qualitative modeling of kinematic robots. In *18th International Workshop on Qualitative Reasoning*, Illinois, USA, 2004.

[LRL+97] Hector Levesque, Raymond Reiter, Yves Lespérance, Fangzhen Lin, and Richard B. Scherl. GOLOG: A logic programming language for dynamic domains. *Journal of Logic Programming*, 31:59–84, 1997.

[May90] Peter S. Maybeck. The Kalman filter, An introduction to concepts. In *Autonomous Robot Vehicles*, pages 194–204, 1990.

[MH96] Robin R. Murphy and David Hershberger. Classifying and recovering from sensing failures in autonomous mobile robots. In *AAAI/IAAI, Vol. 2*, pages 922–929, 1996.

[Min88] Michel Minoux. LTUR: A Simplified Linear-time Unit Resolution Algorithm for Horn Formulae and Computer Implementation. *Information Processing Letters*, 29:1–12, 1988.

[MLM02] Wolfang Maass, Robert A. Legenstein, and Henry Markram. A new approach towards vision suggested by biologically realistic neural microcircuit models. In H. H. Buelthoff, S. W. Lee, T. A. Poggio, and C. Wallraven, editors, *Biologically Motivated Computer Vision. Proc. of the Second International Workshop, BMCV 2002*, volume 2525 of *Lecture Notes in Computer Science*, pages 282–293. Springer (Berlin), 2002.

[MNM02] Wolfgang Maass, Thomas Natschlaeger, and Henry Markram. Real-time computing without stable states: A new framework for neural computation based on perturbations. *Neural Computation*, 14(11):2531–2560, 2002.

[MNPW98] Nicola Muscettola, P. Pandurang Nayak, Barney Pell, and Brian C. Williams. Remote agent: To boldly go where no AI system has gone before. *Artificial Intelligence*, 103(1-2):5–48, August 1998.

[MP97] Thomas L. McCluskey and J. M. Porteous. Engineering and compiling planning domain models to promote validity and efficiency. *Artificial Intelligence*, 95(1):1–65, 1997.

[MRT03] Michael Montemerlo, Nicholas N. Roy, and Sebastian Thrun. Perspectives on standardization in mobile robot programming: The carnegie mellon navigation (CAR-

MEN) toolkit. In *Proceedings of the Conference on Intelligent Robots and Systems (IROS)*, 2003.

[MS96] Andreas Malik and Peter Struss. Diagnosis of dynamic systems does not necessarily require simulation. In *Proceedings of the Seventh International Workshop on Principles of Diagnosis*, 1996.

[MSS95] Andreas Malik, Peter Struss, and Martin Sachenbacher. Qualitative modeling is the key – a successful feasibility study in automated generation of diagnosis guidelines and failuer mode and effects analysis for mechatronic car subsystems. In *Proceedings of the Sixth International Workshop on Principles of Diagnosis*, 1995.

[Mül00] Klaus Müller. Roboterfußball: Multiagentensystem. Master's thesis, Universität Freiburg, 2000. In German.

[Mur02] Robin R. Murphy. *Introduction to AI Robotics*. MIT Press, 2002.

[MWT98] Henry Markram, Yun Wang, and Misha Tsodyks. Differential signaling via the same axon of neocortical pyramidal neurons. *PNAS*, 95(9):5323–5328, 1998.

[Nil84] Nils J. Nilsson. Shakey the Robot. Technical Report 325, SRI International, Menlo Park, CA, 1984.

[Nil94] Nils J. Nilsson. Teleo-reactive programs for agent control. *Journal of Artificial Intelligence Research*, 1:139–158, 1994.

[OC03] Anders Orebäck and Henrik I. Christensen. Evaluation of Architectures for Mobile Robotics. *Autonomous Robots*, 14:33–49, 2003.

[Ore04] Anders Orebäck. *A Component Framework for Autonomous Mobile Robots*. PhD thesis, KTH Numerical and Computer Science, 2004.

[PAC04] Vasco Pires, Miguel Arroz, and Luis Custodio. Logic Based Hybrid Decision System for a Multi-robot Team. In *Proceedings of the 8th Conference on Intelligent Autonomous Systems*, 2004.

[Pea95] Barak A. Pearlmutter. Gradient calculation for dynamic recurrent neural networks: A survey. *IEEE Transactions on Neural Networks*, 6(5):1212–1228, 1995.

[PW92] J. Scott Penberthy and Daniel S. Weld. UCPOP: A sound, complete, partial order planner for ADL. In Bernhard Nebel, Charles Rich, and William Swartout, editors, *KR'92. Principles of Knowledge Representation and Reasoning: Proceedings of the*

Third International Conference, pages 103–114. Morgan Kaufmann, San Mateo, California, 1992.

[Ree99] Christian Reetz. Aktionsauswahl in dynamischen Umgebungen am Beispiel Roboterfußball. Master's thesis, Universität Freiburg, 1999. In German.

[Rei87] Raymond Reiter. A theory of diagnosis from first principles. *Artificial Intelligence*, 32(1):57–95, 1987.

[RH01] Jussi Rintanen and Jörg Hoffmann. An overview of recent algorithms for AI planning. *KI*, 15(2):5–11, 2001.

[Rin00] Jussi Rintanen. An iterative algorithm for synthesizing invariants. In *AAAI/IAAI*, pages 806–811, 2000.

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

[RSB98] Stergios I. Roumeliotis, Gaurav S. Sukhatme, and George A. Bekey. Sensor fault detection and identification in a mobile robot. In *IEEE Conf on Intelligent Robots and Systems*, pages 1383 – 1388, Victoria, Canada, 1998.

[SA98] Reid Simmons and David Apfelbaum. A task description language for robot control. In *Proceedings of Conference on Intelligent Robotics and Systems*, October 1998. Vancouver, Canada.

[SBB+06] Gerald Steinbauer, Mathias Brandstötter, Martin Buchleitner, Stefan Galler, Simon Jantscher, Gerald Krammer, Martin Mörth, Jörg Weber, and Martin Weiglhofer. Mostly Harmless Team Description 2006 - Robust Control of Mobile Robots. In *Proceedings of the RoboCup International Symposium*, Bremen, Germany, 2006.

[SFF+03] Gerald Steinbauer, Michael Faschinger, Gordon Fraser, Arndt Mühlenfeld, Stefan Richter, Gernot Wöber, and Jürgen Wolf. Mostly Harmless Team Description. In *Proceedings of the International RoboCup Symposium*, 2003.

[SGH+97] Reid Simmons, Richard Goodwin, Karen Zita Haigh, Sven Koenig, and Joseph O'Sullivan. A layered architecture for office delivery robots. In W. Lewis Johnson and Barbara Hayes-Roth, editors, *Proceedings of the First International Conference on Autonomous Agents (Agents'97)*, pages 245–252, New York, 5–8, 1997. ACM Press.

[Sim94] Reid Simmons. Structured control for autonomous robots. In *IEEE Transactions of Robotics and Automation*, volume 10, pages 34–43, 1994.

[SKGB04] Freek Stulp, Alexandra Kirsch, Suat Gedikli, and Michael Beetz. Agilo robocuppers 2004. In *RoboCup International Symposium*, 2004.

[SMW05] Gerald Steinbauer, Martin Mörth, and Franz Wotawa. Real-time diagnosis and repair of faults of robot control software. In *RoboCup International Symposium*, Osaka, Japan, 2005.

[SS01] Martin Sachenbacher and Peter Struss. AQUA: A Framework for Automated Qualitative Abstraction. In *Working Papers of the 15th International Workshop on Qualitative Reasoning (QR-01)*, 2001.

[SW05a] Gerald Steinbauer and Franz Wotawa. Challenges in runtime detecting and locating faults in autonomous mobile robots. In *Working Notes of the IJCAI-05 Workshop on Model-Based Systems*, Edinburgh, Scotland, 2005.

[SW05b] Gerald Steinbauer and Franz Wotawa. Detecting and locating faults in the control software of autonomous mobile robots. In *16th International Workshop on Principles of Diagnosis (DX-05)*, pages 13–18, Monetrey, USA, 2005.

[SW05c] Gerald Steinbauer and Franz Wotawa. Detecting and locating faults in the control software of autonomous mobile robots. In *19th International Joint Conference on Artificial Intelligence (IJCAI-05)*, Edinburgh, UK, 2005.

[SWW05] Gerald Steinbauer, Jörg Weber, and Franz Wotawa. From the real-world to its qualitative representation – practical lessons learned. In *18th International Workshop on Qualitative Reasoning (QR-05)*, pages 186–191, Graz, Austria, 2005.

[Tea05] The Miro Developer Team. *Miro Manual*. Department of Computer Science, University of Ulm, 0.9.4 edition, 2005.

[TWWB02] Alex M. Thomson, David C. West, Yun Wang, and Peter Bannister. Synaptic connections and small circuits involving excitatory and inhibitory neurons in layers 2-5 of adult rat and cat neocortex: Triple intracellular recordings and biocytin labelling in vitro. *Cerebral Cortex*, 12(9):936–953, 2002.

[USEK02] Hans Utz, Stefan Sablatng, Stefan Enderle, and Gerhard K. Kraetzschmar. Miro – middleware for mobile robot applications. *IEEE Transactions on Robotics and Automation, Special Issue on Object-Oriented Distributed Control Architectures*, 18(4):493–497, August 2002.

[USM05] Hans Utz, Freek Stulp, and Arndt Mühlenfeld. Sharing belief in teams of heterogeneous robots. In D. Nardi, Riedmiller, Sammut M., and J. C., Santos-Victor, editors, *RoboCup 2004: Robot Soccer World Cup VIII*, volume 3276 of *Lecture Notes in Artificial Intelligence*, pages 508–515, Berlin, Heidelberg, Germany, 2005. Springer-Verlag.

[Utz05] Hans Utz. *Advanced Software Concepts and Technologies for Autonomous Mobile Robotics*. PhD thesis, University of Ulm, Neuroinformatics, 2005.

[VGST04] Vandi Verma, Geoffry Gordon, Reid Simmons, and Sebastioan Thrun. Real-time fault diagnosis. *IEEE Robotics & Automation Magazine*, 11(2):56 – 66, 2004.

[Wel99] Daniel S. Weld. Recent advances in ai planning. *AI Magazine*, 20(2):93–123, 1999.