REACTIVE TASK EXECUTION OF A MOBILE ROBOT

JUKKA RIEKKI

Infotech Oulu

OULU 1999

JUKKA RIEKKI

REACTIVE TASK EXECUTION OF A MOBILE ROBOT

Academic Dissertation to be presented with the assent of the Faculty of technology, University of Oulu, for public discussion in Raahensali (Auditorium L 10), Linnanmaa, on december 18th, 1998, at 12 noon.

OULUN YLIOPISTO, OULU 1999

Manuscript received 24.11.1998
Accepted 30.11.1998

Communicated by
Professor Matti Juhala
Doctor Alex Zelinsky

ISBN 951-42-5131-8
(URL: http://herkules.oulu.fi/isbn9514251318/)

ALSO AVAILABLE IN PRINTED FORMAT

ISBN 951-42-5099-0
ISSN 0355-3213 (URL: http://herkules.oulu.fi/issn03553213/)

OULU UNIVERSITY LIBRARY
OULU 1999

To my family

Riekki, Jukka, Reactive task execution of a mobile robot
Infotech Oulu and Department of Electrical Engineering, University of Oulu, PL 4500, FIN-90401 Oulu, Finland
1998
Oulu, Finland
(Manuscript received 25 November 1998)

Abstract

This thesis presents a novel control architecture, Samba, for reactive task execution. Reactive task execution implies goal-oriented and reactive properties from a robot and the ability to execute several tasks at the same time, also in a dynamic environment.

These requirements are fulfilled in Samba by the representations of goals, intermediate results, and robot actions. The key idea in Samba is to produce continuously reactions for all the important objects in the environment. These reactions are represented as action maps, which are a novel representation for robot actions. An action map specifies for each possible action how preferable the action is from the perspective of the producer of the map. The preferences are shown by assigning a weight to each action.

Tasks are executed by modifying and combining action maps. The tasks can be either reasoned by a higher layer or triggered by sensor data. Action maps, and the methods for modifying and combining them, enable executing tasks in parallel and considering the dynamics of the environment. Further, as the action maps are produced continuously from sensor data, the robot actions are based on the current state of the environment.

Markers describe goals and intermediate results. They facilitate managing the complexity of the system. Markers describing intermediate results decompose the system vertically, into producers and consumers of data. Markers describing goals decompose the control system horizontally, into a Samba layer and a higher layer of reasoning tasks. Tasks flow via markers from the higher layer to the Samba layer.

Markers are tested on a real robot equipped with a stereo gaze platform. Further, the Samba architecture is applied to playing soccer. Experiments were carried out in the 1997 and 1998 RoboCup competitions. These experiments show that the Samba architecture is a potential alternative for controlling a mobile robot in a dynamic environment.

Keywords: control architecture, behavior-based, marker, action map

Acknowledgments

This study was carried out in the Machine Vision and Media Processing Group at the Department of Electrical Engineering of the University of Oulu, Finland and the Electrotechnical Laboratory, Tsukuba, Japan during the years 1993–1998.

If I could thank only one person here, I would select without any doubt my supervisor Professor Juha Röning. He gave me enough time to carry out the research; to nurture the vague ideas of the control architecture until they filled the pages of this book. My deepest thanks to him. Luckily, I am not too constrained in giving the credits, so here come the rest.

I am grateful to Professor Matti Pietikäinen for allowing me to work in the Machine Vision and Media Processing Group for the numerous years this work lasted. Furthermore, I wish to thank my colleagues in the Machine Vision and Media Processing Group including Visa Koivunen, Janne Heikkilä, Hannu Kauppinen, Lasse Jyrkinen, Janne Haverinen, Tapio Repo, Marko Laakso, and Jukka Kontinen for creating a pleasant atmosphere. I am also grateful to Jussi Pajala and Antti Tikanmäki for their hard work in making Samba tick.

I would like to express my gratitude to Dr. Yasuo Kuniyoshi for inviting me to work with him at the Electrotechnical Laboratory during the years 1994–1996. Yasuo’s great ideas and enthusiasm had a major influence on this thesis. I still miss the fruitful discussions with Yasuo, Alex Zelinsky, Sebastien Rougeaux, Luc Berthouse, Nitin Afzulpurkar, Paul Bakker, Kenji Konaka, Nobuyuki Kita, Polly Pook, and many others. I also had wonderful time with them in the legendary and numerous parties at the ETL. Also the parties and other activities with the great number of foreigners living in Tsukuba were enjoyable. My special thanks to Maija and Timo Ropponen who introduced Tsukuba to my wife and myself when we had just arrived there and took us to many interesting places and events.

Professor Matti Juhala from the Helsinki University of Technology and Alex Zelinsky from the Australian National University are acknowledged for reviewing and commenting on the thesis. Their constructive criticism improved the quality of the manuscript considerably. Many thanks also to Leon Trembath for reading and commenting on the thesis.

This work was financially supported by the Academy of Finland, the Tauno Tönning Foundation, the Science Foundation of Seppo Säynäjäkangas, the Emil Aaltonen Foundation, and the Japanese Government. Their support is gratefully acknowledged.

I am deeply grateful to my mother Lea and father Pekka for their love and care over the years. Most of all, I want to thank my dear wife Maritta for her patience and understanding and our sons, Niklas and Ville, for putting robotics research in its right perspective.

Oulu, November 25, 1998 Jukka Riekki

Contents

1. Introduction

1.1 Background

Although robots have shown their usefulness in industrial applications, a mobile robot operating in our everyday environment is a rare sight. This is because the complex and dynamic environment and tasks of reasonable complexity together set such strict requirements for both the hardware and software components of the robot, especially for a mobile and autonomous robot.

A robot needs sophisticated sensors and sensor data processing routines for creating and updating a rich description of the environment. Complex tasks call for powerful reasoning methods. In a dynamic and imperfectly predictable environment the robot has to integrate task execution with fast reactions to unexpected situations. All this requires a considerable amount of processing power, particularly because the robot has to operate at the pace of the environment. Furthermore, moving in our everyday environment and handling objects designed for humans demands advanced actuator mechanisms and control methods. Mobility in turn constrains the weight, size, and power consumption of the hardware components. Finally, autonomous operation requires that the robot adapts to the environment, learns new environments, and learns new skills.

In this thesis, we concentrate on reactive task execution – on integrating task execution with fast reactions to unexpected situations. Of course, robots also need the other capabilities listed above, and robots with such capabilities have to be affordable before we can expect them to occupy our daily environment. However, the capability of reactive task execution alone would increase the performance of the robots in many applications.

Reactive task execution implies both goal-oriented and reactive properties from a robot. Being goal-oriented, the robot reasons its actions based on given goals. Being reactive, the robot reacts quickly to unexpected events in the environment while performing these actions. The ability to execute several tasks at the same time is essential in reactive task execution. This is because the robot should be capable of reacting to unexpected events simultaneously with achieving a goal.[1] Also complex goals as such can require the

1 When reacting to an unexpected event, the event determines a goal to be achieved. As a task is a piece of work to be done and a goal is the object of this effort (Oxford 1985), it follows that both the given goals and the goals triggered by the unexpected events are achieved by executing tasks.

achieving of several subgoals simultaneously. Executing several tasks in parallel is challenging, especially in a dynamic environment where each task might require consideration of different aspects of the environment changing over time.

This work is about control architectures of mobile robots. Control architecture specifies the structure of a control system: how the system is decomposed into computational modules, and how the modules communicate with each other. In other words, control architecture is the blueprint of a control system. Although the hardware also sets constraints on the skills of the robot, it is the architecture that determines whether the robot is reactive, goal-oriented, or both.

Goal-oriented control architecture is based on the deliberative approach. A goal is achieved by acquiring a model of the environment, planning task execution based on the model, a description of the goal, and a description of the possible actions, and commanding the robot based on the plan. Weld (1994) gives an introduction to planning. All control systems based on the deliberative approach share this sense-model-plan-act framework. Solving problems in symbolic, abstracted domains is characteristic of the deliberative approach.

The deliberative approach has the advantage that the robot is able to reason the actions needed to execute a wide variety of complex tasks. The ability to execute several tasks at the same time depends on the representation of time utilized in the planner. Actions must not be treated as instantaneous, and overlapping actions must be allowed. Chapman (1987) lists planners fulfilling these requirements.

The main shortcoming of a deliberative system is that it is not reactive. First, planning requires a complete model of the environment. Our everyday environment contains such a vast number of features that the sensing and model updating operations required by a complete model cannot be performed in real-time. Actually, the situation is worse: a consistent model can be maintained only in trivial cases (Gini 1988). This problem of maintaining the proper correspondence between the symbols and reality has been labeled the symbol grounding problem (Malcolm & Smithers 1990).

Second, the amount of computation required by planning would prevent fast reactions to environment events, even if a complete model were available. Chapman presents a theorem about planners telling us that “no upper bound can be put on the amount of time required to solve a problem” (1987, 344). Although the theorem was later criticized for being too pessimistic (Weld 1994), it nevertheless argues for the difficulty of this problem.

Similar criticism against the deliberative approach has been presented by Gat (1993). He lists the time required by planning, the level of completeness and fidelity a planner requires from a world model, and the rate of changes in the world as the main reasons causing a planner to fail in robot control.

Replacing the planner and the model with a hierarchy of simpler planners and models speeds up the reactions of a deliberative system, as the amount of model updating and planning between perception and action decreases. However, it is questionable whether this approach can produce the reaction times needed at the lower levels of the control hierarchy. As a model and a planner (or several of them) are still on the signal path, the related problems remain.

Around 1984, a number of people started to develop alternatives for the deliberative approach (Brooks 1986a, Agre & Chapman 1990, Kaelbling & Rosenschein 1990). The resulting approach has been labeled the situated approach, where the term “situated” emphasizes that the robots are situated in the world, they do not deal with abstract descriptions (Brooks 1991a). The situated control systems are also called behavior-based (or behavioristic), emphasizing the interaction of the robot with the world. Although Mataric (1997) distinguishes reactive systems from behavior-based ones, we use the terms “reactive”, “behavior-based”, and “situated” interchangeably. This is because, from the perspective of this work, fast reactions are the most distinctive property of all the situated approaches.

The subsumption architecture suggested by Brooks (1986a) is the best known of the behavior-based control architectures. In a behavior-based control system, robot actions are calculated directly from sensor data.[2] The system consists of a set of behaviors operating in parallel. Each behavior knows how to react to an event or situation in the environment. Each behavior receives only the sensor data needed to detect the event or situation and to calculate a response to it. Thus, the amount of sensor data processing is minimized. Furthermore, a behavior sends commands directly to an actuator. A task is executed by coordinating the behaviors; by selecting at each moment the behavior (behaviors) that is (are) allowed to control the robot. This type of a control system has the obvious advantage of operating at the pace of the environment – it is able to react to the unexpected events in the environment in a timely fashion.

The small amount of memory and simplicity of representations are fundamental features of the behavior-based architecture. There is no world model and the representations of the messages flowing between the modules of the system are simple. These features facilitate fast reactions, as no complex transformations between representations nor expensive model updating are needed.

To be able to achieve a wide variety of goals, behavior-based control architecture must have a method for goal-oriented behavior coordination. This method is used to select and combine at each moment the set of behaviors that cause progress for the task at hand. The behavior coordination method determines the number of tasks that can be executed at the same time, that is, the number of different behaviors that can control the robot in parallel. To perform several different tasks at the same time, commands produced by the behaviors must be combined instead of selecting a single behavior to control the robot. The coordination method also constrains the ability to take the dynamics of the environment into account, especially when several tasks need to be performed in parallel. In such a case, each behavior considers individually how the dynamics of the environment affect the execution of the corresponding task. The method of combining the behaviors defines how successfully these considerations are preserved in the combined command.

The simplicity of the representations and small amount of memory hinder implementing a versatile behavior coordination method. The simplicity of the representation describing robot actions causes a loss of information on the signal path. This constrains the set of behavior coordination methods that can be implemented. Further, modules storing goals would facilitate activating the right set of behaviors to achieve a goal.

2 We use the terms “action” and “command” frequently. They are defined as follows: a command is an order given by one in authority and an action is something performed (Webster 1996). Thus, a behavior produces commands and the execution of a command produces an action. As the terms have such a close relationship, we use them interchangeably.

Many researchers have built behavior-based control systems for controlling mobile robots. Brooks and his colleagues have reported an impressive list of behavior-based mobile robots (Brooks 1989b). Other researchers have either based their systems on the subsumption architecture or developed their own reactive architectures (Agre & Chapman 1990, Mataric 1997, Rosenblatt & Payton 1989).

Malcolm and Smithers suggest that the combination of the deliberative and reactive approaches, a hybrid architecture could be the most elegant and simple way of building complex systems (Malcolm & Smithers 1990). In a hybrid control system, a reactive system reacts to unexpected events and executes the tasks given by the deliberative system. In the terms of Malcolm and Smithers, the reactive subsystem hosts a virtual machine interface described by the commands it is able to execute. Hybrid architecture has also been defended by Gat (1993), who claims that most of the problems associated with models of the environment (in his terms, stored internal state) can be solved by maintaining models only at a high level of abstraction. Hybrid architectures have been suggested also by several other researchers (Payton 1986, Payton *et al.* 1990, Arkin 1990, Ferguson 1994).

Regardless of whether the requirements for reactive task execution are to be fulfilled by a purely reactive approach or by a hybrid approach, a reactive system has to be made more deliberative. In both cases, the reactive architecture must contain a method for goal-oriented behavior coordination and allow executing several tasks in parallel, also in a dynamic environment. In the case of a hybrid system, the deliberative system utilizes the goal-oriented behavior coordination method in controlling the reactive system. In the case of a purely reactive system, the system components utilizing the coordination method have a closer resemblance to the behaviors being controlled.

The requirements for reactive task execution cannot be fulfilled with reasonable effort in the reported reactive architectures. The main reason is the characteristic limitations of behavior-based architecture that hinder implementing a versatile behavior coordination method. Although improved behavior coordination methods have been suggested (e.g. Rosenblatt & Payton 1989), sufficient capability to execute tasks in parallel in a dynamic environment has not yet been achieved. Hence, more work needs to be done before reactive task execution is realized.

1.2 The scope of the thesis

The purpose of this thesis is to develop control architecture for a mobile robot operating in our daily environment. The robot moves in the environment and executes simple object manipulation tasks. The robot is equipped with suitable actuators and sensors producing rich information about the environment. The emphasis is on interacting with the environment rather than planning complex sequences of actions. The environment consists of static objects at fixed locations, of movable objects, and other agents (humans and robots). The other agents either participate in the task execution or just need to be avoided.

The environment is man-made and dynamic. It is not changed because of the robot. For example, there are no artificial landmarks in the environment. This guarantees the widest set of potential applications. Further, the robot does not have any accurate global reference frame. As there are no artificial landmarks, there is no reasonable way to update accurate locations in an artificial reference frame.

In this work, we develop the control architecture for reactive task execution by extending a reactive, behavior-based control architecture. The approach of extending behavior-based architecture was chosen because basing robot actions on the environment – situatedness – was seen as the most important property of a control system. The actions have to be at each moment sensible when interpreted according to the current state of the environment. We do not exclude the deliberative approach. Although the emphasis is on the lower levels of the control hierarchy, we also define an interface to a higher level system that produces goals for the lower levels. Whether this higher level system consists of a planner and a model or of behaviors is left open.

We do not develop a formal theory in this thesis. Developing control architecture requires deep insight into the problem of controlling a mobile robot – insight that can be achieved only through experiments. For this reason, we concentrate on implementing the control architecture and making experiments. The experiments help us to gain a better insight into the problem, so we can improve the architecture, implement it, and make more experiments. This loop is repeated until the architecture is mature enough to be formalized. This pattern in AI research has been justified by Chapman as follows (1987, 334):

> "When working at the frontiers of knowledge, you will make no progress if you wait to proceed until you understand clearly what you are doing. But it is also hard to know where to go next before the scruffy work is neatened up."

Chapman calls the stage of gaining insight "scruffy" and the following stage of defining a formal theory "neat". As the implementing and testing of control architecture are such time consuming tasks, we have not yet reached the "neat" stage.

Also Malcolm and Smithers support this research approach. They argue that since we as yet know so little about designing and building artificial creatures, we should use the simplest possible experimental domains which focus on the important problems, and which test our assumptions as soon as possible (Malcolm & Smithers 1990).

1.3 The contribution of the thesis

The contributions of this thesis lie in developing novel architecture for reactive task execution. A new representation for robot actions, the action map representation, is the central contribution. Action maps and the suggested method for combining them enable several tasks to be executed in parallel without coding explicitly each different possible combination of parallel tasks. A further contribution is the confirmation on the importance of a robot being situated in the world instead of dealing with abstract descriptions.

In this thesis, we propose reactive task execution to be the main requirement for a mobile robot operating in our daily environment. We define a general behavior-based architecture and analyze the requirements for reactive task execution. Further, we evaluate the suitability of the existing architectures for reactive task execution. The result of the evaluation is that fulfilling the requirements for reactive task execution is difficult with existing architectures.

We suggest that the requirements for reactive task execution can be fulfilled by a hybrid system consisting of a reactive and a task layer. The Samba architecture developed in this thesis forms the reactive layer of such a system. A Samba control system produces reactions to the unexpected events in the environment and performs the tasks reasoned by the task layer.

The Samba architecture is behavior-based extended with markers, action maps and goal-oriented behavior coordination. Markers are simple processing elements that ground task related data on sensor data flow and communicate it to behaviors. They describe intermediate results and goals and form interfaces. As a result, markers help to manage complex systems. We suggest also a new method, the Maximum of Absolute Values method, for combining the actions produced by the behaviors. This method is based on a new representation of robot actions, the action map representation. The representation enables several tasks to be executed at the same time. A further advantage is that it facilitates taking environment dynamics into account.

The developed goal-oriented behavior coordination method is built on markers. A task marker activates the set of behaviors performing a task. The task markers form an interface between the reactive and the task layer, as the tasks are reasoned by the task layer. Markers are utilized also in representing the state of the environment. This approach does not suffer from the symbol grounding problem, as all data is grounded physically. Further, the system operates in real time, as only the data related to the current task is stored in the markers and thus the updating is fast. Finally, markers and action maps enable a new type of decomposition into producers and consumers of primitive reactions.

In addition to the Samba architecture, this work resulted also in insights confirming the importance of situatedness advocated by Brooks (1991a). First, experiments about markers taught us the utmost importance of grounding everything physically. Second, the experiments about action maps suggest that planning optimal action sequences is rarely necessary as generating plausible actions directly based on sensor data produces such good results.

This work started in cooperation with Dr. Kuniyoshi (Kuniyoshi et al. 1994). Dr. Kuniyoshi was inspired by Chapman’s Sonja architecture and suggested markers for mobile robot control (Chapman 1991). The author specified, at the Electrotechnical Laboratory in Dr. Kuniyoshi’s group, the first version of the control architecture, implemented it, and tested the architecture with real robots. The author did not participate in implementing the image processing nor low-level control routines. At the University of Oulu, the author developed the markers further and added the action map representation into the architecture. Further, the author, together with research assistants, implemented a control system based on the Samba architecture to illustrate the properties of the architecture.

In both the Samba architecture and the Sonja architecture markers serve as an interface between the producers and consumers of data. In Samba, however, markers also form an interface between a system producing tasks and a system performing them. The action map representation resembles the one used in the DAMN architecture (Rosenblatt & Payton 1989). The main difference is that in the DAMN architecture, speed and direction commands are produced separately, whereas an action map representing velocities contains both the speed and the direction. Further, the weights in an action map are based directly on the situation around the robot. The votes in the DAMN architecture have no such direct relation to the situation.

1.4 The outline of the thesis

We start the thesis by defining in the second chapter a control architecture for reactive task execution in general terms. The definition serves as a foundation for the architecture developed later in this thesis. In the second chapter, we also analyze the requirements of reactive task execution in detail and survey the related work. The emphasis of the survey is on reactive architectures and how they fulfill the requirements for reactive task execution.

The third chapter describes the Samba control architecture. This chapter contains the main contribution of the thesis. We start by describing the key concepts of the Samba control architecture and illustrating the architecture with an example. Then we specify the architecture in detail. Furthermore, we discuss the features of the Samba architecture.

The fourth and fifth chapters cover the experiments. We describe the implemented control systems and the experiments performed. In the experiments presented in the fourth chapter, a mobile robot follows other robots and removes the obstacles of their trajectories. The control system controls a mobile robot equipped with a stereo gaze platform. In the experiments presented in the fifth chapter, a team of 11 simulated mobile robots plays soccer against another simulated robot team. Each robot is controlled by an independent control system. In the both sets of experiments, the primary goal is to verify that markers and action maps facilitate fulfilling the requirements for reactive task execution. Furthermore, the aim is to prove that these extensions do not destroy the reactiveness of a behavior-based system. A secondary goal is to illustrate the properties of the control architecture.

Finally, the sixth chapter contains the discussion and the seventh chapter the conclusions.

2. Architecture for reactive task execution

The aim of this chapter is to define an architecture for reactive task execution in general terms, and to present the related work. The architecture defined in this chapter serves as a foundation for the architecture developed in this thesis.

In the introduction, we specified that reactive task execution requires both goal-oriented and reactive properties and the ability to execute several tasks at the same time, also in a dynamic environment. In this chapter, we discuss how these requirements can be fulfilled. We also consider methods for managing the complexity of the control system. As this work is about control architectures, the emphasis is on the modules of the architecture and on the representations.

In the first section, we define a general reactive, behavior-based architecture. The architecture is inspired by the subsumption architecture proposed by Brooks (1986a). We classify the modules of the architecture into sensor, behavior, arbiter, and actuator modules. Further, we discuss the behavior coordination problem in detail and define a more versatile behavior coordination method than the one used in the subsumption architecture. This behavior coordination method allows several tasks to be executed simultaneously.

In the second section, we explain how the goal-oriented properties can be achieved by connecting a higher level system, task layer, to the behavior-based system (i.e. reactive layer). The task layer reasons the goals and communicates them to the reactive layer. Hence, the reactive layer has to be able to coordinate the behaviors based on goals, to perform goal-oriented behavior coordination. In this section, we list the requirements set by the goal-oriented behavior coordination and discuss how they can be fulfilled. Specifically, we describe how modules storing goals facilitate goal-oriented behavior coordination. In both the first and second sections, we discuss how the representation of robot commands affects the behavior coordination.

In the third section, we discuss modules storing intermediate results as a tool for managing the complexity of a control system. In both the second and third sections, we also discuss how the new features affect the reactive properties of the general behavior-based architecture. We survey the related work in all three sections. The emphasis is on the reactive architectures and how they fulfill the requirements for reactive task execution.

The organization of this chapter was inspired by Tsotsos (1995). He argues that behavior-based architecture does not scale to large problems. The main reason is the lack of representations: there are no representations for intermediate results nor representations describing goals explicitly. In this chapter, we present how to remove this shortcoming with modules storing goals and intermediate results.

2.1 Reactive control architecture

A control system based on reactive control architecture can be defined as follows:

A ***reactive control system*** *is capable of reacting to unexpected events at the pace of the environment.*

An ***event*** *is a change in the state of the environment that implies a reaction from the system.*

An ***unexpected event*** *is an event that was not foreseen when reasoning the action that is under execution when the event occurs.*

The most important property of a reactive control system is its fast reactions. Fast reactions imply short signal paths from perception to action and small amounts of computation in the modules on the signal paths. The subsumption architecture suggested by Brooks (1986a) fulfills these requirements. In the following section, we explain this architecture and some other reactive architectures. We also survey the arguments for and against these architectures and, more generally, for and against the reactive approach. The following two sections define a general behavior-based architecture.

2.1.1 Subsumption architecture and other reactive architectures

In 1986, Brooks proposed decomposing the mobile robot control problem based on task-achieving behaviors into horizontal control layers (1986a). The behavioristic decomposition is shown in Fig. 1, together with the traditional, deliberative decomposition of a mobile robot control system.

Brooks named the resulting architecture the subsumption architecture. In this architecture, a control layer is built from a set of small modules that send messages to each other. Each module is a finite state machine augmented with registers and timers (Brooks 1989a). A behavior emerges as the collaboration of a number of modules. Each higher level layer defines a more specific class of behaviors. A control layer is permitted to examine data from the lower layers and to inject data into the lower layers suppressing the normal data flow. The name “subsumption architecture” is drawn from this property of a control layer subsuming the roles of lower level layers.

A central feature of the subsumption architecture is that the control system can be built incrementally – the control system is operational as soon as the first, lowest layer has been

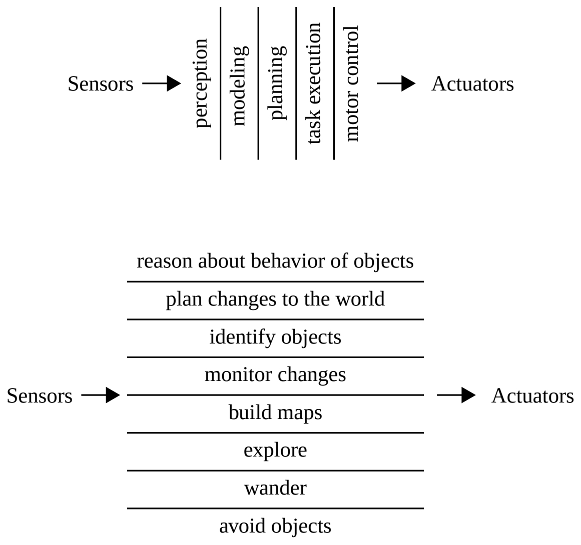


Fig. 1. The difference between the deliberative and the behavioristic decomposition of a mobile robot control system (adapted from Brooks 1986a, Fig. 1 and Fig. 2). Top: deliberative decomposition into functional units. A functional unit receives data from its predecessor and sends data to its successor. The functional units form together a signal path from perception to action. Bottom: behavioristic decomposition into control layers. Each control layer contains a complete signal path from perception to action.

finished. Brooks argues that this way of incrementally building intelligence is needed because it is not yet known how to build a complete, truly intelligent system (1986b).

Other central features of the subsumption architecture are the small amount of memory and simplicity of representations. There is no world model and the representations of the messages flowing between the modules are simple. In the Behavioral Language that is used to program control systems based on the subsumption architecture, there are no shared data structures across behaviors and registers holding messages are usually no more than eight bits wide (Brooks 1990). Even one bit data paths have been used in some implementations (Brooks 1987). These features facilitate fast reactions, as no complex transformations between representations are needed and no expensive model updating nor planning operations based on the model are performed.

Numerous control systems have been built based on the subsumption architecture. In Brooks' laboratory at MIT, researchers have built more than ten subsumption robots (Brooks 1989b). These MIT Mobile Robots include, among other things, Allen who heads towards distant places and avoids static and dynamic obstacles, Herbert who steals empty soda cans from people's desks, and the six legged robot Genghis who follows people.

Agre and Chapman started to doubt the deliberative approach at the same time as Brooks. They coined the term "the situated approach" for this new approach (Agre & Chapman 1995, Agre & Chapman 1990, Chapman 1991). This approach emphasizes that a central feature of all activity is that it takes place in some specific, ongoing situation. The

Pengi and Sonja control systems implemented by Agre and Chapman register continuously the important aspects of the ongoing situation and act on them. The control systems are composed of peripheral systems responsible for perception and effector control, and of central systems responsible for registering and acting on aspects of the situation. The central systems are made entirely of combinational logic. The central system of the Sonja architecture consists of registrars computing aspects of the situation, proposers suggesting actions based on the aspects, and arbiters choosing between conflicting actions.

Agre and Chapman also developed a representation agreeing with the situated approach. An aspect of the ongoing situation is a deictic representation.[3] A deictic representation represents things in terms of their relationship with the agent. The central idea in this representation is to describe only the things that are causally connected to the agent (indexicality) and are relevant to the task at hand (functionality). Chapman (1991, 30) presents the following example of a deictic representation: *the-cup-I-am-drinking-from* is an entity, a thing in a particular relationship to the agent, and *the-cup-I-am-drinking-from-is-almost-empty* is an aspect of that entity.

Kaelbling and Rosenschein (1990) suggested the situated automata approach for controlling robots. Brooks mentions them as the pioneers of the reactive approach (Brooks 1991b). A situated automata maps situations in the environment into actions of the robot, via the internal state of the robot. For each situation in a specified domain, the automata contains a condition and the corresponding action. The automata is executed at a constant frequency. At each cycle, the conditions are evaluated in a predefined order and the action corresponding with the first found satisfied condition is executed. The automata is created by a compiler, which receives as input a declarative specification of the agent’s top-level goal and a set of goal-reduction rules.

The universal planner suggested by Schoppers (1987) resembles the situated automata approach. We classify both the situated automata and universal planning approaches as reactive ones, because no planning is employed on the signal path. An appropriate action is specified (i.e. planned) off-line to every possible situation within a given domain. At the execution time, the actual situation is classified and response planned for that situation is performed. The planner is rather a tool for creating the reactive control system than a component of a deliberative control system.

Many other researchers have either based their systems on the subsumption architecture or developed their own situated architectures (Connell 1987, Rosenblatt & Payton 1989, Malcolm & Smithers 1990, Mataric 1997).

The subsumption architecture, or, more generally the reactive approach has provoked strong arguments both for and against in the AI and robotics communities. The central argument against the reactive approach is that it does not scale to complex problems because the amount of modules and connections between them grow too fast with the complexity of the domain (Tsotsos 1995). Tsotsos reached this conclusion by utilizing a strict interpretation of the behavioristic approach; by rejecting all hierarchical computations and representations describing goals and intermediate results explicitly. However, the strictness of interpretation is not the main point. Although Brooks rejects explicit representations of goals within the control system (1991b), the control systems

3 In earlier papers the deictic representation was called as the indexical-functional representation.

based on the subsumption architecture contain intermediate representations (e.g. the sonar module in Fig. 5 in 1986a). Further, the layers of the subsumption architecture are hierarchical layers.[4] Hence, although the strict behaviorist position has its value in provoking the AI community to reconsider the deliberative approach of building intelligent robots, it should not be taken as a list of requirements for reactive architectures.

Similar arguments with Tsotsos have been presented by Ginsberg (1989a). He claims that the size of a reactive system grows exponentially with the complexity of the domain. Although the primary target of this argument was the universal planning approach (Schoppers 1987)[5], Ginsberg claimed that it was valid also for some other reactive systems such as the situated approach of Agre and Chapman. Chapman (1989) responded by pointing out that this argument does not hold for the situated approach, because it exploits the structure of the task and a small amount of state to reduce the computation's size.

The main argument for the behavioristic approach is that the control system must be connected to the real world; a robot must interact with the real world from the very beginning. In other words, a robot must be situated and embodied. Situatedness means that the robot deals with the here and now of the world instead of abstract descriptions. Embodiment means that the robot has a body and experiences the world directly. Brooks claims that the deliberative agents fulfill neither of these requirements; they are neither situated nor embodied.

2.1.2 General behavior-based architecture

We define in this chapter a general behavior-based architecture containing behavior, sensor, actuator, and arbiter modules. All modules of the architecture have the structure shown in Fig. 2. A module consists of input buffers, output buffers, memory, and functions. A module receives signals into its input buffers, calculates output, places it into output buffers and sends it forward. The module uses the data in the memory when calculating output and can also modify the data during the calculations. The output signals flow through wires to input buffers of other modules.

A sensor module receives data from physical sensors, processes it, and sends it forward. A virtual sensor module processes data produced by other modules; it does not receive data from a physical sensor. An actuator module receives commands from behaviors and sends them to a physical actuator. The arbiter module is defined later in this chapter. The fourth type of a module, a behavior module, transforms sensor data into commands for an actuator. The behavior and the goal of a behavior are defined as follows:

> ***A behavior** transforms sensory data continuously into commands decreasing the distance between the current state of the environment and a goal state.*

4 Brooks rejects hierarchical computation in which a process calls another as a subroutine. In the subsumption architecture, the results of a lower layer are available when required, that is, the higher layer is not required to call the lower layer to get the results.

5 See also Schoppers' defense of universal planners (Schoppers 1989) and Ginsberg's answer (Ginsberg 1989b).

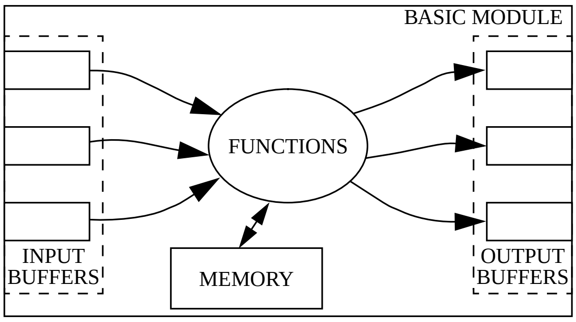


Fig. 2. The basic module of a behavior-based system.

The goal of a behavior *is a set of environment states called goal states.*

The distance between the current state of the environment and a goal state refers to the distance in the state space. A behavior does not have to measure this distance. It suffices to determine the direction of the goal state and to produce a command causing a state transition to that direction.

Behaviors are individually encapsulated as behavior modules. In the original subsumption architecture, a different approach was utilized: behaviors emerged as the collaboration of a number of simple state machines (Brooks 1986a). However, in the later versions behaviors are defined explicitly. For example, in the Behavioral Language (Brooks 1990) behaviors are defined as rule sets which are then compiled into a set of finite state machines.

The definition of a behavior presented above differs from some of the definitions suggested by others. Mataric (1997) employs behaviors as perceptual filters. For example, a “corridor-finding” behavior notifies other behaviors when it is confident that the robot is in a corridor.[6] In the general behavior-based architecture, modules that do not produce commands are not behaviors, as they do not have a direct effect on the observed behavior of the robot.

The first requirement for fast reactions, short signal paths, is fulfilled in a behavior-based system by specializing each signal path from perception to action into achieving a simple goal. A signal path consists of a sensor module, a behavior module, and an actuator module (Fig. 3).

The second requirement, a small amount of computation, is fulfilled by keeping the functions and the representations simple. The functions of the modules are simple, as a sensor module extracts from sensor data only the data needed to calculate that single response. The behavior module receives a minimum amount of data and does a minimum

6 However, Mataric presents in the same paper also the definition “a control law that satisfies a set of constraints to achieve and maintain a particular goal” which resembles closely the definition given in this work.

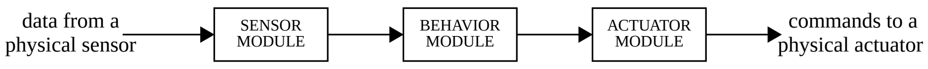


Fig. 3. A basic signal path of a behavior-based system. This path from perception to action is specialized into achieving a simple goal.

amount of computation. The actuator module just commands the actuators. In other words, all modules at a signal path process only the data related to a single goal. Simple representations keep the transformations between representations simple. For example, the subsumption architecture described in the previous section allows only single numbers to flow between the modules.

The amount of computation can also be decreased by minimizing the amount of information stored by a behavior. In this work, no specific constraints are set on the functions, representations, or amount of information. The general constraints are that behaviors do not employ centralized representations and that they operate at the pace of the environment.

A behavior is activated by an activating event. The calculation continues – the behavior is active – until another predefined event (deactivating event) deactivates the behavior. Also the elapsing of a certain amount of time can be interpreted as an event, as time passes in the environment. Another perspective to behavior activation is that each behavior has a set of environment states (valid states) in which it is active. A behavior becomes active (i.e. valid) when an activating event causes a state transition into a valid state. The behavior remains active until a deactivating event causes a state transition into a non-valid state. The set of valid states and the activating and deactivating events are illustrated in Fig. 4. Also

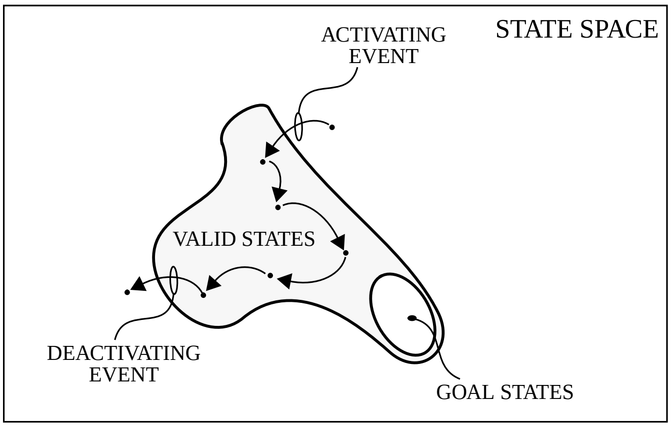


Fig. 4. The set of valid states, the set of goal states, and the activating and deactivating events of a behavior. The dots mark environment states and arrows transitions between them (that is, events).

the goal states of a behavior are shown in Fig. 4. In the situation described in the figure, the behavior produces commands in the valid area of the state space but it does not succeed in controlling the environment into a goal state.

As an example of behavior activation, an obstacle avoidance behavior is considered. The behavior is activated when the robot perceives an obstacle nearer than a safety limit. Hence, the valid states of the behavior are the states in which the obstacle is too near the robot. The goal states are the states in which the distance between the robot and the obstacle exceeds the safety limit. Reaching a goal state deactivates the behavior.

Another example is a behavior driving the robot towards a goal. The goal is a certain object or feature in the environment. The behavior is active when the robot is too far from the goal. The goal states are the states in which the robot is near enough the goal location. The event deactivating the behavior is the reaching of the goal with the required accuracy. The behavior can alternatively keep the robot at the goal once it has reached it.

2.1.3 Behavior coordination

As a behavior achieves a simple goal, a control system of one behavior is of no use. A behavior-based system performing simple tasks can easily contain tens of behaviors.[7] When the sets of valid states of behaviors intersect, several behaviors produce commands to an actuator at the same time. That is, there are environment states in which several behaviors produce commands, each to achieve a different goal. For example, one behavior might calculate how to reach a goal and another how to avoid an obstacle. When both a goal and an obstacle are being perceived, both behaviors are active.

Thus, we need a method for coordinating the behaviors. Another alternative would be to specify non-intersecting valid states for all behaviors. However, this would violate the modularity of the system. When implementing a behavior, we would have to consider the valid states of all the other behaviors. This is a questionable method, especially in a large system, as it introduces hidden dependencies between the modules. When there is a large number of such dependencies, it is difficult to manage changes to the system. It is much easier to specify each behavior separately – considering only the states in which the behavior in question should be active – and solve the conflicts between the behaviors separately. Hence, behavior coordination is unavoidable in all systems larger than a couple of behaviors.

Coordination can be defined as follows:

> ***Coordination** is the process of guiding several entities somehow to work towards a common goal. This may include preventing some entities from working.*

The goals of the coordinated behaviors are subgoals of the common goal.

7 The largest behavior-based control system known by the author is the control system for a rough terrain locomotion built by Ferrell (1995). This system contains approximately 1500 concurrently running processes. Behaviors and other modules are combinations of these processes. Ferrell does not mention the number of behaviors in the system.

When several behaviors produce commands (are active) at the same time, coordination means that a subset of the commands is selected and the selected commands are combined in some way. This is done by the fourth type of module, an arbiter module. Fig. 5 shows a system containing several behaviors and an arbiter. An arbiter extends the signal path from perception to action, but as long as arbitration is a simple process, it has no observable effect on the response times of the system.

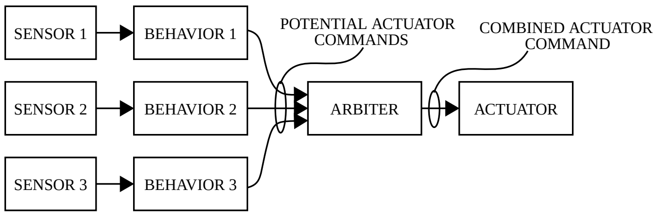


Fig. 5. A behavior-based system.

The simplest arbitration method is to define the priorities of the behaviors beforehand. In this arbitration method, *the method of static priorities*, an arbiter selects at each moment the behavior among the active behaviors that has the highest priority. The priorities are static, as they are determined when building the system. This method is used in subsumption architecture, although arbitration is performed by suppress nodes instead of arbiter modules (Brooks 1986a). A suppress node receives commands from two behaviors and passes through the command of the higher priority behavior when the behaviors are active at the same time. More than two behaviors are arbitrated in several stages by several suppress nodes. A situated automata (Kaelbling & Rosenschein 1990) performs similar arbitration, although it does not contain behaviors. A situated automata contains a condition-action pair for each situation in a specified domain. The automata evaluates the conditions in a predefined order and executes the action corresponding to the first found satisfied condition. Payton (1986) has also reported an architecture utilizing the method of static priorities.

The obvious disadvantage of the method of static priorities is that the priorities cannot be changed dynamically based on the environment state. *The method of dynamic priorities* does not suffer from this disadvantage. In this method, the arbiter modifies the priorities based on both the state of the control system and the state of the environment. Dynamic arbitration is utilized in the Sonja architecture (Chapman 1991).

In both the static and dynamic methods, the robot commands are represented as a single set of parameters required by the corresponding actuator. For example, for an actuator moving the robot base, a command might contain a new direction relative to the old one and a new driving speed. Alternatively, direction and speed commands might be sent separately.

In an even more versatile and complex arbitration approach, the arbiter – instead of selecting a single high priority behavior – combines the commands of all the active behaviors. This arbitration method, *the method of combining*, produces compromises between the behaviors. This gives the system the ability to satisfy several behaviors, i.e., to execute tasks in parallel.

Arkin has proposed that commands could be combined by summing vectors (Arkin 1990). A driving command is represented as a velocity vector indicating the direction and speed at which the robot is to move. For example, when an obstacle avoidance and a goal approaching behavior produce commands at the same time, summing these two vectors and normalizing the sum vector produces a command that causes the robot to both avoid the obstacle and approach the goal. Also Mataric (1997) arbitrated behaviors in the Nerd Herd experiments by summing velocity vectors.

When the vector summing method is utilized, it cannot be guaranteed that the combined command does satisfy any of the active behaviors. This hinders executing tasks in parallel. This disadvantage is a consequence of the fact that each behavior sends forward only the one command producing the maximum satisfaction for that behavior. The summing operation results in a command that in most cases does not equal any of the suggested commands. Satisfaction cannot be guaranteed, as it is not known how big variations each behavior allows to the command it suggests. In other words, the representation of robot commands causes a loss of information on the signal path.

If the behaviors would send forward also alternative commands that do not give the maximum satisfaction yet satisfy them, the arbiter could combine the commands in such a way that the resulting command satisfies several behaviors. Of course, the amount of satisfied behaviors depends on the variations the behaviors allow the optimal commands. The smaller the set of alternative commands each behavior accepts, the higher the probability that the sets do not intersect. However, at least one behavior would always be satisfied. Furthermore, if the behaviors would define the level of satisfaction for each alternative command, the arbiter could maximize the satisfaction of the set of active behaviors.

Rosenblatt has proposed a representation for describing alternative commands (Rosenblatt & Payton 1989, Rosenblatt & Thorpe 1995). In his DAMN architecture, behaviors produce votes for each different possible command parameter value. A command is represented as a set of (parameter value, vote) -pairs. The arbiter sums the votes sent by different behaviors and selects the parameter value with the maximum sum of votes. The speed and turn commands are processed separately, i.e., a behavior sends, as one signal, votes for each possible driving speed and as another signal votes for each possible turning angle.

The three methods of arbitration are gathered in Table 1.

The arbitration methods can also be compared by analyzing how they modify the sets of valid states of two behaviors when the sets intersect. In the static method, the arbiter subtracts the intersecting valid states from the set of the behavior with the lower priority. In the dynamic method, the arbiter divides the intersecting valid states between the two behaviors. In the combining method, the valid states of the behaviors stay intact. The effect a behavior has on the resulting command depends both on the information the behavior sends to the arbiter and on the tuning of the arbiter. These differences between the arbitration methods are illustrated in Fig. 6.

As the arbiter receives commands from all behaviors connected to an actuator, it can select the best behavior at each moment. However, the best behavior is not necessarily the optimal one. This is because the information about the environment and about the

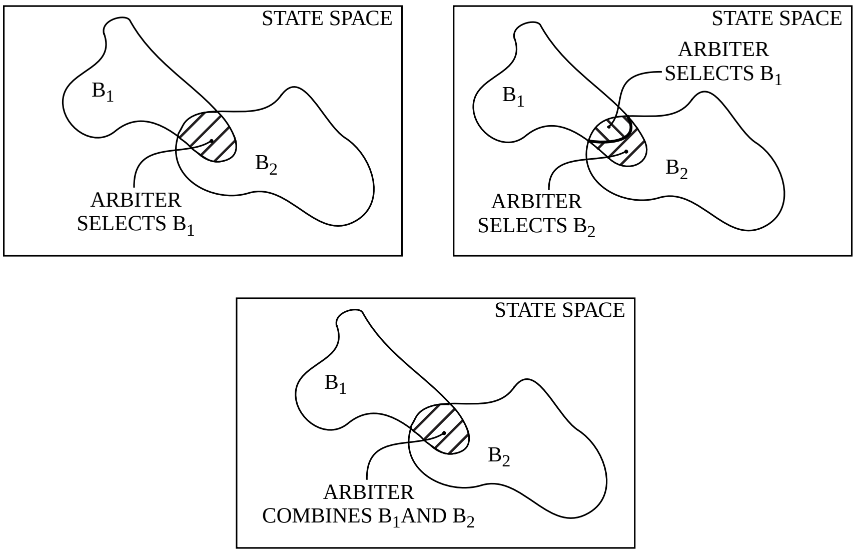


Fig. 6. The operation of an arbiter when the valid states of two behaviors intersect. Top left: static priorities, behavior B_1 has higher priority than behavior B_2. Top right: dynamic priorities. Bottom: combining.

Table 1. The three arbitration methods.

Static priorities	Definition: Select the behavior among the active behaviors that has the highest predetermined priority. Result: One behavior is satisfied, the others fail.
Dynamic priorities	Definition: Calculate priorities for the behaviors based on the state of the control system and the environment. Select the behavior among the active behaviors that has the highest priority. Result: One behavior is satisfied, the others fail.
Combining	Definition: Combine the commands of the active behaviors. Result: The amount of satisfaction depends on the variations the behaviors allow to the optimal commands and on the representation of robot actions. The amount of satisfied behaviors varies between one and the amount of active behaviors.

commands satisfying the behaviors is incomplete. The information about the commands is incomplete because information is lost on the signal path. The amount of information that is lost is determined by the representation of robot commands. Hence, the representation of robot commands is a central element of behavior coordination. It determines whether the arbiter is able to produce commands that satisfy several behaviors in parallel. In other words, the representation of robot commands determines whether the requirement set for reactive task execution, the ability to execute several tasks in parallel, can be fulfilled.

When the method of static priorities is utilized in arbitration, tasks cannot be executed in parallel by combining the commands produced by independent behaviors. Instead, a new behavior must be implemented for each different combination of parallel tasks. The method of dynamic priorities has the same shortcoming.When the method of combining is utilized, the ability to execute tasks in parallel depends on the representation of commands. From the representations described above, the one utilized in the DAMN architecture (Rosenblatt & Thorpe 1995) enables the widest variety of arbitration methods, as it was designed to minimize the loss of information on the signal path. However, even this method has difficulties in satisfying several behaviors in parallel, especially in a dynamic environment, and just because of loss of information. Satisfying several behaviors in parallel is more challenging in a dynamic environment because in addition to being at the right location, the robot has to be there at the right moment.

Information is lost in the DAMN architecture, because turn and speed commands are processed separately. To be more accurate, as the turn and speed commands are sent as separate signals, information about which direction and speed form a velocity satisfying a behavior is lost. This impedes guaranteeing that the direction and speed selected by the arbiter are reasoned by the same behavior. For example, a control system might consist of a Catch behavior catching a moving target and an Avoid behavior avoiding a moving obstacle. When the behaviors are active in parallel, the arbiter selects the direction with the maximum sum of votes and the speed with the maximum sum of votes. Both behaviors give high votes for the (direction, speed) pairs corresponding to the velocities that will result in the robot being at the right location at the right time. However, as turn and speed commands are sent as separate signals, the correspondence between them is easily broken. The arbiter can select, for example, a direction satisfying the Catch behavior and a speed satisfying the Avoid behavior. When this happens, both behaviors may fail.

As a conclusion, to allow several tasks to be executed in parallel, also in a dynamic environment, behaviors should produce alternative satisfying commands. Further, the direction and speed suggested by a behavior should be integrated into a single command to prevent the correspondence between them being broken.

As mentioned above, in addition to the information loss on the signal path, incomplete information about the environment degrades the optimality of the system operation. The information is incomplete, as the system does not contain models of the environment and the sensors are imperfect. The sensors produce data only about the local environment and they measure only a limited set of the features in the local environment. Further, the measurements contain noise.

The information loss on the signal path and incomplete information about the environment make it difficult to guarantee that the robot will succeed in all situations. As an example, a control system might consist of two behaviors, DriveStraight and AvoidObstacles. The DriveStraight behavior interprets the location of the goal as an attractive force and produces a vector pointing towards the goal. The AvoidObstacles behavior interprets the obstacles as repulsing forces and produces a vector pointing away from the obstacles. When these two vectors are combined utilizing the vector summing method, the robot stops when the vectors cancel each other, even though the robot has not reached the goal. This type of location is called as the local minimum.

As another example, a control system consisting of DriveStraight and FollowWall behaviors could be considered. The behaviors suggest driving directions for the robot. The DriveStraight behavior always suggests driving straight towards the goal. The FollowWall behavior suggests following the wall always when a wall has been perceived (to keep the example simple, all obstacles are interpreted as walls). The DriveStraight behavior is the only valid behavior when there are no walls inside the wall sensing radius. Both behaviors are valid when a wall which is to be followed is being perceived.

Clearly, when arbitration is based on static priorities, the FollowWall behavior must have a higher priority than the DriveStraight behavior, as otherwise the robot would collide into a wall when the wall is between the robot and the goal. However, if we set the priority of the FollowWall to be the higher one, the robot can reach only goals near a wall once it starts to follow it.

This problem can be solved by calculating the priorities dynamically. Then, the DriveStraight behavior can have a lower priority when a wall is being perceived, except when the goal is inside the wall sensing radius and is not blocked by a wall. However, also the dynamic arbitration fails when the region traversable by the robot is not simply connected. Once the robot starts to follow a wall surrounding a hole in the region, it never stops if the goal stays outside the wall sensing radius.

Furthermore, even if the traversable region is simply connected, once the robot starts to follow a wall it cannot reach goals that stay outside the wall sensing radius. This problem cannot be solved by relaxing the conditions for the DriveStraight behavior. If the DriveStraight behavior would be selected when no wall is perceived in the direction of the goal, the robot could be caught in a loop. This could happen when a wall is located between the robot and the goal but further than the wall sensing radius.

Another interesting approach would be to use static priorities and have a third behavior, ApproachGoal, that would have the highest priority. This behavior would drive the robot straight towards the goal when the goal is inside the wall sensing radius and not blocked by a wall. Thus, in this situation, dynamic arbitration can be avoided by rethinking the decomposition of the total robot behavior into individual behaviors.

These examples show the difficulty of behavior coordination. No matter how carefully the arbiter has been designed, it is difficult to guarantee that there will not be more complex situations in which the arbitration fails. This problem could be relieved by learning during operation the places and how to travel between them (Mataric 1997, Zelinsky 1996). For example, the robot could learn the sequences of DriveStraight and FollowWall activations needed to drive from one place to another. But even when the robot would learn its environment, the problem of behavior coordination would be encountered when avoiding unexpected obstacles and when exploring new areas.

2.2 Goal-oriented behavior coordination

To be suitable for reactive task execution, a behavior-based control architecture must have a method for goal-oriented behavior coordination. This type of coordination is needed because a goal-oriented system is by definition capable of achieving a wide variety of goals. Some of these goals can be achieved by a single behavior, but more complex goals require a coordinated operation of several behaviors. Although also the general behavior-based system has its own goals, goal-oriented behavior coordination is utilized only in coordinating based on goals given by the task layer. This is because the goals of the general behavior-based system are represented implicitly inside behaviors.

Goal-oriented behavior coordination can be divided into two phases. First, the set of behaviors required in achieving the goal are activated. If several goals need to be achieved in parallel, several behavior sets are activated. Some behaviors can also be disabled. In addition to activating the right set of behaviors, the parameters of the behaviors must also be set based on the goals. The same mechanism can be used both to activate the behaviors and to set their parameters. Second, the most important behaviors are selected from the active ones and the commands produced by them are combined. This selection and combination is performed by an arbiter, which uses all the available information about the current environment state and the goals. The combined command is then executed. We use the terms activation and arbitration, respectively, from these two types of coordination. These terms, together with other terms used in the discussion, are specified below:

A ***goal*** *is a set of environment states. The robot is to achieve one of these states by changing the environment with its actuators.*

A ***goal-oriented system*** *is capable of achieving a wide variety of goals. The system reasons its actions based on the specified goals and the state of the environment. The goals can be either reasoned by the system itself or given by other systems.*

Goal-oriented behavior coordination *is coordinating behaviors based on the goals of the system.*

Activation *is a behavior coordination method in which the set of behaviors required in achieving a goal are activated.*

Arbitration *is a behavior coordination method in which the most important behaviors are selected from the active ones and the commands produced by them are combined.*

A goal is defined as a set of environment states instead of a single state. This is because in most cases a small set of features of the environment determine a goal. The values of other features do not matter. Furthermore, the exact value of a feature might not matter as long as the value is in a given range.

Coordination by activation requires a mechanism for activating the proper set of behaviors. This requirement can be achieved by decomposing the system into subsystems achieving goals. A subsystem contains all the modules participating in achieving the goal.

In the simplest case, a subsystem consists of one behavior. Each subsystem has an interface module – a goal module – for receiving a goal for the subsystem. The goal module activates all behaviors in the subsystem when it receives an activation signal specifying a goal. A goal module extends the signal path from perception to action, but as long as it performs only simple operations, it has no observable effect on the response times of the system.

To keep the system situated, also goal modules need to be grounded to the environment. This is achieved by associating a goal module with an object in the environment. The main components of a goal module are a location and a task specification. The task specification defines the task and the role of the object in that task. A task specification does not need to be explicit but it can be encoded in the identity of a goal module. This type of a goal module resembles the aspects of the Pengi system (Agre & Chapman 1995) which are discussed in more detail in Section 2.3.

The central advantage of goal modules is that they facilitate managing the complexity of a control system. Goal modules allow sets of behaviors to be activated and disabled by a single command and assist in hiding details inside subsystems. Both the details about modules and about their connections can be hidden. This facilitates considerably the implementation and modification of a control system, as the programmer does not have to understand every last detail of the system in order to modify it.

A goal module contains memory for storing goal parameters. In terms of Chapman (1989), goal modules add a small amount of state in the architecture and also allow abstraction and modularity. These are the properties that, according to Chapman, are required to reduce a reactive system's size to a feasible level.

The advantages discussed above could be achieved without storing goals, by activating the modules of a subsystem directly from the goal signals sent to the subsystem. However, storing the goals at the interface increases the modularity. Particularly when the subsystem reasoning goals (the producer) and the subsystem achieving goals (the consumer) operate at different rates, a goal module produces a cleaner interface and simplifies the system. The producer usually operates at a lower rate than the consumer. Without a goal module, either the producer would need to send the goal signals at the rate of the consumer, or the consumer would need to store the goal by other means, separately for each module of the consumer. This would be required as the modules need to have the goal parameters available also during the execution cycles when the goal signal is not received from the producer. Sending goal signals at the rate of the consumer is clearly out of question if the producer, by its very nature, operates at a lower rate. Storing goal parameters separately for each module would be inefficient as each set of parameters would have to be updated separately based on sensor data (as they have to be grounded).

The arbitration in goal-oriented behavior coordination resembles the arbitration in the behavior-based architecture described in the previous section. The only difference is that in goal-oriented coordination the arbiter also uses data produced by the goal modules in the arbitration process. Arbitration, like activation, facilitates managing the complexity of a system. This property of arbitration was discussed at the beginning of Section 2.1.3.

Activation and arbitration exclude each other to some extent. If only one behavior is activated at each moment, there is no need for arbitration. On the other hand, if an arbiter is able to select the most important behavior from among all behaviors, there is no need for activation. However, in goal-oriented systems both types of coordination are needed.

Using both activation and arbitration produces simpler systems than using either activation or arbitration alone. This is because activation reduces the amount of arbitration and vice versa.

Activation and arbitration in a goal-oriented reactive system are illustrated in Fig. 7. The task layer produces goals for the general behavior-based system. To keep the figure clear, all behaviors of a subsystem send signals to the same actuator. They could also command different actuators.

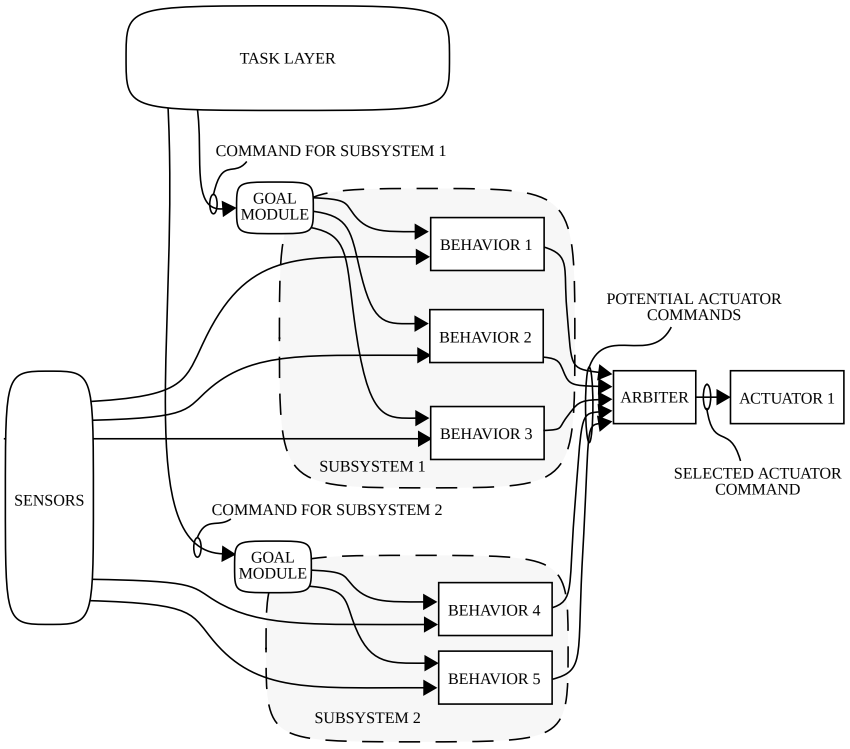


Fig. 7. A goal-oriented reactive system. In addition to the signals drawn in the figure, modules can receive signals from sensors and goal modules.

In the Sonja architecture, a subsystem is called an abstract operation (Chapman 1991). An abstract operation is defined in terms of the buses transferring inputs to the subsystem. This is achieved by specifying these buses as the ports of the subsystem. A port is connected to all modules that have theories about what the corresponding input should be. The values sent to a port are arbitrated by an arbiter. All modules of the subsystem can read the arbitrated values. Although Chapman does not define an abstract operation as a goal-achieving subsystem, he uses it as such. For example, various modules send goals for the abstract operation *navigate*. The difference between a subsystem in the general behavior-based architecture and an abstract operation is that an abstract operation does not contain a module which stores inputs.

Brooks rejects explicit representations of goals (1991b). In the subsumption architecture, the message sent by a module selecting a goal has no implicit semantics – its meaning is designed separately into the sender and each receiver. Further, the module selecting a goal sends the message separately to each module using the goal in its computations. This approach has the disadvantage that the system becomes difficult to manage when the number of modules increases. As connectivity is not constrained, it becomes difficult to understand how the system works and how it should be modified in order to change the behavior of the robot in some specified way. To manage the complexity we need constraints on the connectivity. Goal modules bring us these constraints.

The arbitration method of the DAMN architecture (Rosenblatt 1995) can be utilized in goal-oriented behavior coordination to arbitrate the activated behaviors. This representation of robot commands facilitates satisfying several behaviors in parallel. However, the DAMN architecture defines no activation method.

The goal-oriented behavior coordination method suggested above has similarities with the Alliance architecture (Parker 1994). The motivational behaviors in Alliance and the goal modules in the suggested method activate and deactivate groups of behaviors. The main difference is that the goal modules also send data to the behaviors, whereas the motivational behaviors merely enable other behaviors. Furthermore, the behaviors belonging to a subsystem receive a signal from a goal module, whereas in Alliance the motivational behaviors control the outputs of the behaviors.

Coordination methods activating and deactivating groups of behaviors are common in hybrid systems. In ATLANTIS, this type of coordination is performed by a sequencer that coordinates the behaviors in the reactive system based on directions from the deliberative system (Gat 1993). Payton (1986) proposes composing activation sets from subsets of the total collection of behaviors. During operation, only the behaviors from the current activation set are enabled to produce commands. Hence, this approach resembles the approach of Parker.

2.3 Intermediate results

Implementing a complex behavior-based system is difficult without modules storing intermediate results. As the modules form interfaces between the producers and consumers of data, they facilitate decomposing the system into subsystems and hence managing the complexity. These interface modules reduce the number of connections because producers send data only to the interface modules and consumers receive data only from the interface modules. This is shown in Fig. 8. Furthermore, the modules reduce the amount of computation as they calculate the needed values instead of all consumers repeating the same calculations.

The modules storing intermediate results bring the same advantages as the goal modules; they add a small amount of state to the architecture and also allow abstraction and modularity. This is not a surprise, as goals can be interpreted as intermediate results.

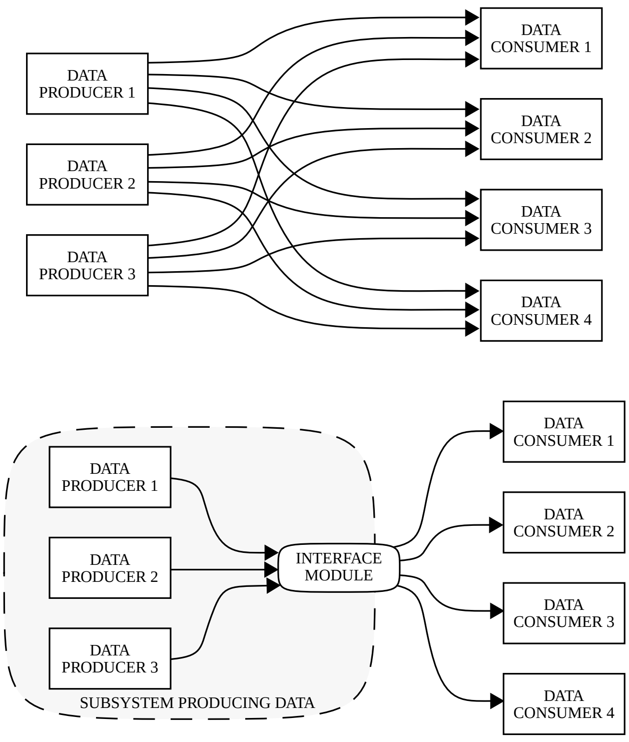


Fig. 8. An example of how decomposing the system reduces the number of connections. Top: every consumer receives all the produced data. Bottom: an interface module receives the produced data, fuses it, and sends it further.

But are all modules storing intermediate results goal modules? For example, if a module storing the color of an object triggers a behavior controlling the robot towards all red objects, is that module a goal module? The module is not a goal module, as the goal “drive towards the red objects” is represented inside the behavior.

Since the modules storing intermediate results represent properties of the objects in the environment, they are labeled as object modules. The general-behavior based architecture extended with object modules is shown in Fig. 9.

The main components of an object module are the location of the object and a set of roles for the object. Object modules are updated based on sensor data, so the location must correspond with a recognizable feature in the environment. The module can be interpreted as floating on the top of the feature in the sensor data flow. Object modules extend the signal path from perception to action, but as they mainly perform computations that would otherwise be performed by other modules they should not have an observable effect on the response times of the system.

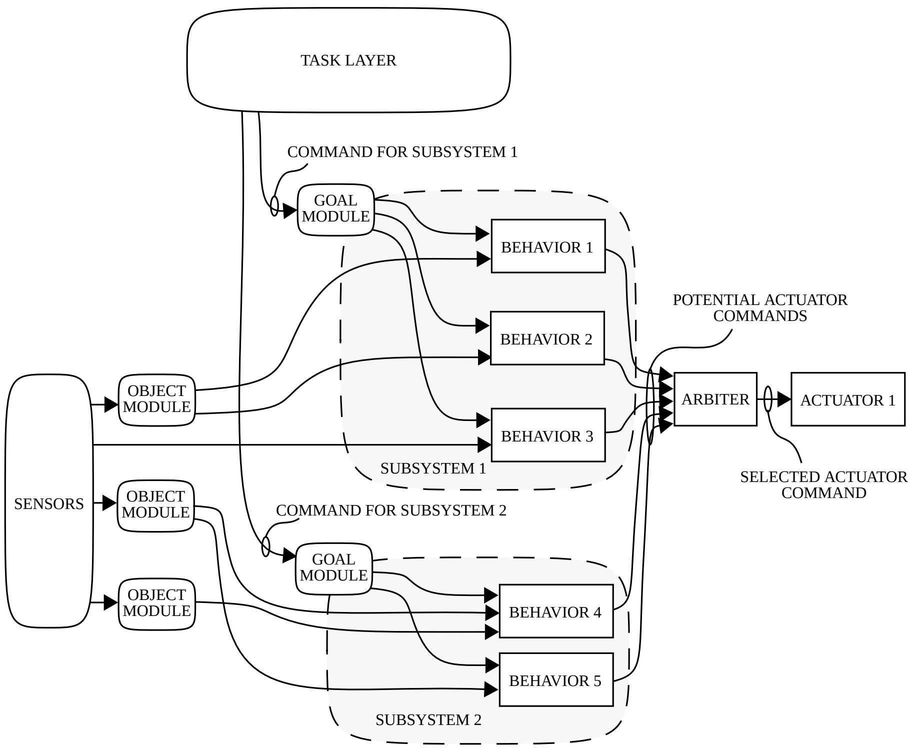


Fig. 9. A general behavior-based system connected to a task layer. In addition to the signals drawn in the figure, modules (and the task layer) can receive signals from sensors, object modules, and goal modules.

For each set of objects that affects the operation of the robot in a specific way there is a unique role. The object sets may intersect, and hence an object can have several roles. The role set of an object module specifies the roles of the corresponding object. For example, if the robot is only capable of avoiding obstacles, all object modules have the role set *{obstacle}*. As another example, a robot might set the table for the agent drinking from a cup in the example presented by Chapman (see page 21). For such a robot, a coffee cup has several roles. In addition to being a coffee cup to be placed at a particular location, the cup also has the role of being a thing on the table, and the role of being an obstacle to be avoided when placing other coffee cups. In this case, the role set of the object module representing a coffee cup is *{coffee-cup, obstacle, a-thing-on-the-table}*.

The identity of the object module determines the most specific role of the object represented by the module. Thus, there are several object modules of the same identity when the robot needs to take into account several objects of a role in parallel. More general roles are defined as unions of the more specific roles. Continuing the above example, the role set of the object module representing a coffee pot is *{coffee-pot, obstacle, a-thing-on-*

the-table}. Hence, a control system for setting a table might contain one *coffee-pot* object module and several *coffee-cup* object modules. In addition, there might be *other-object* modules for all the other things on the table that are always avoided. The union of the roles *coffee-pot*, *coffee-cup,* and *other-object* defines the role *a-thing-on-the-table*. Fig. 10 illustrates these roles.

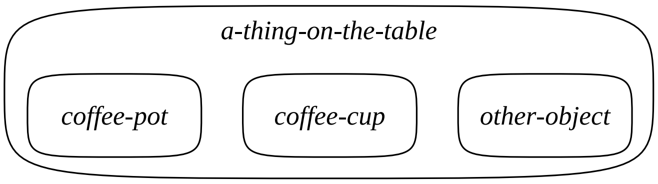


Fig. 10. Classifying the objects a robot setting a table needs to consider. The things on the table are classified into coffee pots, coffee cups, and other objects.

The main purpose of the roles is that they can be used to focus modules on a certain set of objects. For example, only object modules of the role *moving-object* need to be connected to a module calculating distances to moving objects. The role set does not have to be represented explicitly. It suffices to connect each object module of a certain role to each module requiring data about objects of that role. In addition, a module has to decide dynamically which role of the corresponding role set an object module has at each moment. For example, when a module receives data from all object modules representing coffee cups, it has to decide which coffee cup is to be manipulated and which cups are to be avoided. This decision can be based on the data sent by a goal module. The set of cups to be avoided forms a dynamic set of objects having the role *obstacle* together with the other things on the table that are to be avoided.

The role sets depend solely on the purposes of the robot. In other words, the roles classify the objects to the degree of accuracy required by the application. There is a role *moving-object* only if the robot reacts in a special way to moving objects. This approach of attaching a role set to each object module was selected instead of specifying a separate object module for each different role of an object, because it produces less modules, less connections, and less computing. In other words, this approach facilitates managing the complexity of the system.

The subsumption architecture does not contain any specific module type for intermediate results; the modules storing intermediate results are implemented as augmented finite state machines, as are all the other modules. The message has no implicit semantics – its meaning is designed separately into the sender and each receiver. An example of a module storing intermediate results is the Sonar module in the first subsumption system (Brooks 1986a). This module produces a robot centered map of obstacles that is utilized by two other modules, Feelforce and Collide.

The intermediate objects (markers, lines, rays, and activation planes) of the Sonja architecture (Chapman 1991) are intermediate representations. They keep track of intermediate results during sensor data processing and serve as an interface between the sensor data processing part and the rest of the system. Also the object modules serve as an interface between data producers and consumers.

The deictic representations, aspects, in the Pengi (Agre & Chapman 1995) and Sonja architectures are intermediate representations. The aspects are produced by registrars and consumed by the proposers and arbiters. As was discussed earlier in this chapter, the central idea in the deictic representation is to describe only the things that are causally connected to the agent and are relevant to the task at hand. Also the object and goal modules serve the same purpose of constraining the amount of things to be represented. Instead of representing the environment completely in a world model, the objects and features in the neighborhood of the agent that are relevant to the task at hand are represented by these modules. The deictic representation and the one suggested here have two major differences: roles are more general than an aspect and an object module can describe several roles, whereas a registrar produces a single aspect. A role describes a class of objects with a certain property, whereas an aspect is associated both with the circumstances and the purposes of the robot, so it classifies an object more accurately. For example, an object module *cup* represents data about an object belonging to the class of cups, but an aspect *the-cup-I-am-drinking-from-is-almost-empty* determines also the situation and the current activity of the agent. A goal module specifying a task of handling a cup might be *drink-from-this-cup*.

2.4 Discussion

The general behavior-based architecture suggested in this chapter fulfills, together with a task layer, the requirements for reactive task execution. A control system based on this architecture is reactive, as fast reactions are characteristic for a behavior-based system. Goal-oriented operation is achieved with the goal modules and goal-oriented behavior coordination. Further, a representation of robot actions describing alternative actions allows tasks to be executed in parallel, also in a dynamic environment. Finally, modules storing goals and intermediate representations facilitate managing the complexity of the system.

None of the architectures surveyed in this chapter fulfill the requirements for reactive task execution to the same degree. Although it is difficult to understand the detailed operation of these architectures by reading the relatively short papers, it is evident that in each architecture at least one requirement is considerably harder to fulfill than in the architecture suggested here. All the surveyed architectures are reactive. But, in the architectures utilizing either the static or the dynamic method of arbitration, executing tasks in parallel is difficult. Even in the architectures utilizing the method of combining, too much information is lost on signal paths. Furthermore, in architectures lacking modules storing intermediate results and goals, achieving goal-oriented operation and managing the complexity of the system are laborious tasks.

3. Samba

In this chapter, we describe the Samba control architecture. It is based on the architecture defined in Chapter 2. Hence, Samba is a reactive architecture containing modules storing intermediate results and goals. These modules form an interface between data producers and consumers, and an interface to a higher layer system called the task layer. The arbitration method utilized in Samba is that of combining. The representation of robot actions allows several tasks to be executed in parallel, also in a dynamic environment. Hence, together with the task layer, a control system based on the Samba architecture is suitable for reactive task execution.

This chapter is organized as follows. In the first section, we both describe the key concepts of the Samba control architecture and illustrate the architecture by an example. The following sections describe the modules and the representations of the architecture. In the last section, we discuss the features of the architecture.

3.1 Introductory Samba

The Samba architecture contains the following types of modules: Sensors, Actuators, Markers, Behaviors, and Arbiters. All these modules have the structure of the basic module depicted in Fig. 2 and communicate by sending signals to each other. The architecture is shown in Fig. 11. The control system is connected to the external world through Sensors and Actuators. Markers are simple processing elements that ground task related data on sensor data flow and communicate it to behaviors. Behaviors transform the sensor and marker data into commands to actuators. Arbiters select and combine commands. Although a control system can contain more than one arbiter, a system of one arbiter is presented in the following discussion.

Actuators change the local environment as reasoned by the behaviors. An actuator module commands a physical actuator based on the commands sent to it. Sensors produce data about the local environment of the robot. A sensor module receives raw sensor data from a physical sensor. A sensor can also be virtual, in which case it receives data from the other modules and processes it further. The reason for using virtual sensors is twofold. First, the system becomes simpler, as each module needing the information does not have

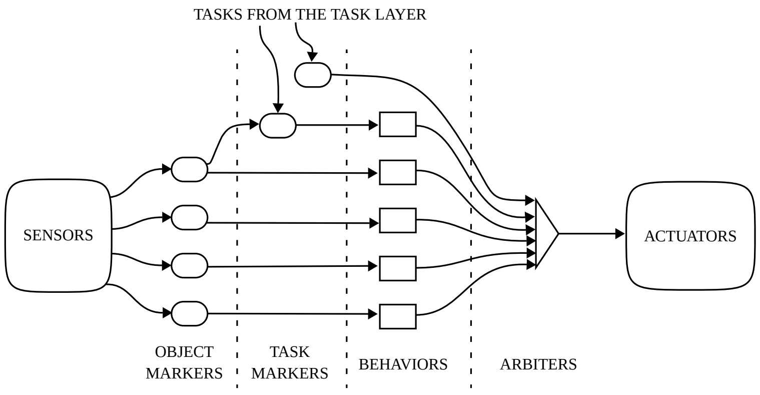


Fig. 11. The Samba architecture. In addition to the signals drawn in the figure, modules can receive signals from sensors and object markers.

to compute it from the raw data. Second, all modules are guaranteed to use the same values.

The local coordinate system of the robot is a polar coordinate system that has its origin at the center of the robot. The main axis of the robot coordinate system points to the direction of the forward motion. The direction of the main axis specifies the zero direction; the angle increases anticlockwise. The robot coordinate system is illustrated in Fig. 12. A polar coordinate system was chosen, because it is precisely the relative directions and distances of objects that are important when the robot chooses its next action. The origin of the coordinate system is bound to the robot for the same reason. Further, the robot-centered coordinate system is a natural way of limiting the amount of data handled by the robot. As the robot moves, the objects further away than the range of the coordinate system are forgotten.

Also objects have local coordinate systems (see Fig. 12). The zero direction of an object coordinate system points to the direction of the main axis of the object. If the main axis cannot be defined based on the observed object features, it coincides with the motion direction of the object. Furthermore, the places (rooms, corridors, halls) have local coordinate systems. The coordinate systems are fixed to natural landmarks.

The robot represents the state of the environment with the object markers. Object markers connect task-related objects to behaviors. They store data about the objects, update the data automatically based on sensor data, and communicate the data to the behaviors. They are modules storing intermediate results. An object marker represents the location, orientation, velocity, and other observable features of an object. The location of an object is defined in the robot coordinate system as a vector (direction, distance). This is the most important piece of information about an object, as objects are indexed by the locations. An object marker has a role set attached to it. The identity of the object marker specifies the most specific role of the object. For each group of objects handled by the robot in a similar manner there is a unique role. The number of object markers depends on the amount of roles. Furthermore, it depends on the number of objects for each role that

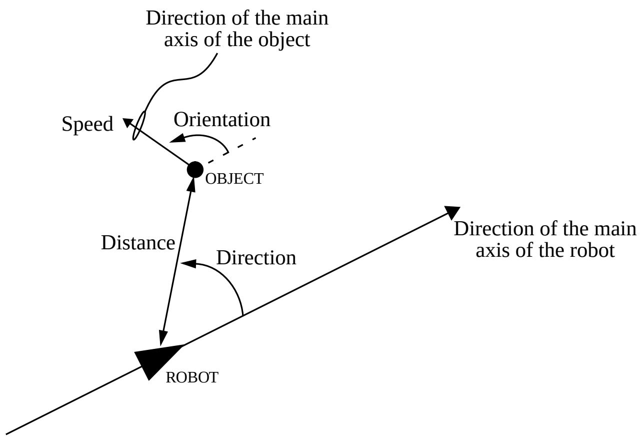

Direction of the main axis of the object
Speed
Orientation
OBJECT
Distance
Direction
Direction of the main axis of the robot
ROBOT

Fig. 12. The coordinate systems of the robot and the objects. The location of the object is represented as a vector (Direction, Distance). The velocity of the object is represented as a vector (Orientation, Speed).

the robot needs to take into account at the same time. For example, if there are roles *stationary-object* and *moving-object* and the robot has to consider simultaneously three stationary and three moving objects, six object markers are needed.

The control system produces both reactions to unexpected events in the environment and the primitive actions required in task execution. It performs tasks by combining these reactions and actions. The tasks can be either given a priori or reasoned by the task layer. The predefined goals are represented implicitly. The tasks reasoned by the task layer are communicated to the Samba control system through task markers. These markers are utilized by the behaviors and the arbiter. Task markers describe goals for the Samba system. Goals are represented as locations and operations the robot should perform at the location. The locations are specified relative to objects. In a similar manner to objects, goals are indexed by their locations. Task markers are updated either by object markers or by the task layer. When updated by object markers, a task marker receives data from the object markers of the role the goal description refers to.

Behaviors produce primitive reactions to objects, such as “go to an object” and “avoid an object”. These reactions are in the form of primitive action maps. A separate primitive action map is calculated for each primitive task of reaching or avoiding an object. Furthermore, a behavior can produce primitive action maps for reaching a location specified by a task marker. A behavior can also utilize representations other than action maps and calculate commands in response to sensor data instead of an object marker.

Tasks that are more complex than reacting to a single object, or reaching a location, are executed by the arbiter. It modifies and combines the primitive action maps into composite action maps based on the task constraints specified by task markers. For the predefined tasks the constraints are represented implicitly inside the arbiter.

An action map specifies for each possible action how preferable the action is from the perspective of the producer of the map. The preferences are shown by assigning a weight to each action. The most common action map type is a velocity map, as the main task of the control system is to decide the velocity of the robot at each moment. Velocity maps are three-dimensional: an action is described by a velocity vector (direction, speed) and for each action there is a weight. The weights form a surface in the polar coordinate system (direction, speed).

A weight for a velocity is calculated based on the time to collision. The shorter the time needed to reach an object, the heavier the weight for that action. For actions that do not result in a collision, the weights are calculated by approximating to what extent the action gives progress to the task of reaching the object. The resulting map contains only positive weights. A map of this kind is called a Goto map, because the performance of the action with the currently heaviest weight on such a map causes the robot to reach the corresponding object. An example of a Goto map is shown in Fig. 13. The map contains a ridge of heavy weights because the object is moving. The robot heading is also shown in the figure. In all figures, action maps are presented at this same orientation.

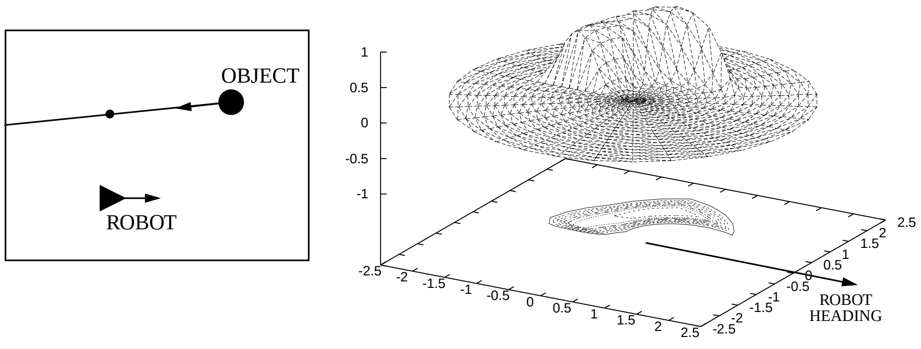


Fig. 13. An example of a Goto map. Left: the situation. Right: the map. If the robot executes the action with the heaviest weight on the Goto map, it reaches the moving object at the location marked with the small dot.

An action map containing negative weights is called an Avoid map. It is calculated either by negating the corresponding Goto map or by a separate procedure. As only positive weights cause actions to be performed, an Avoid map does not alone trigger any motion, but only prevents some actions. In other words, an Avoid map prevents collision with an obstacle when a target is being reached with the help of a Goto map. Fig. 14 shows an Avoid map.

The arbiter combines the Goto and Avoid maps into composite maps by the Maximum of Absolute Values (MAV) method. In this method, the weight with the maximum absolute value is selected for each action. In other words, the shortest time it takes to reach an object is selected for each action and the corresponding weight is stored on the composite map. The sign of a weight on a composite map specifies whether an obstacle or a target would be reached first if a certain action were performed. The absolute value of the weight specifies the time to collision with the object that would be reached first. Fig. 15 illustrates the MAV method for compiling action maps.

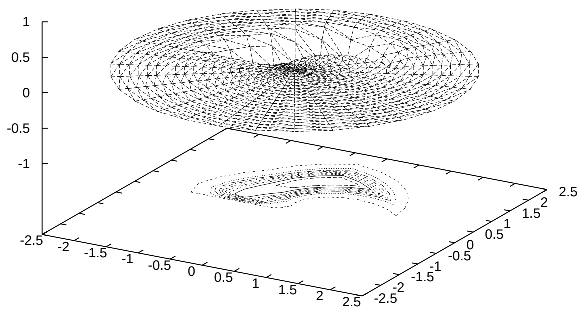


Fig. 14. An Avoid map corresponding to the Goto map shown in Fig. 13.

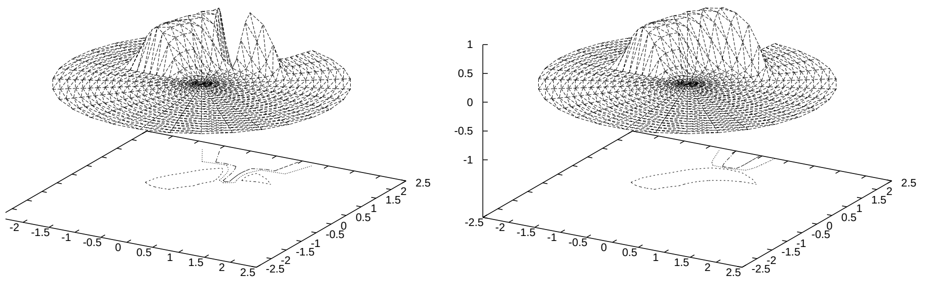


Fig. 15. A Goto map for a moving target and an Avoid map for a stationary obstacle compiled with the MAV method. Left: the obstacle is in front of the trajectory of the target, and actions in that direction are therefore forbidden. Right: the obstacle is behind the trajectory, and hence no actions are forbidden.

A task is performed by sending the action with the heaviest weight in the appropriate composite map to the actuators. The task to be executed at each moment is selected by the arbiter. A simple arbiter combines the composite maps corresponding to the active tasks in a predefined order and selects the best action from the first composite map producing an action good enough. The active tasks at each moment are the predefined tasks and the tasks received from the task markers. A more versatile arbiter considers the current situation when selecting the composite map.

Now, after describing the Samba architecture in a nutshell, we will show a simple example of a Samba control system. Fig. 16 illustrates a control system for controlling the robot towards a specified target while avoiding obstacles.

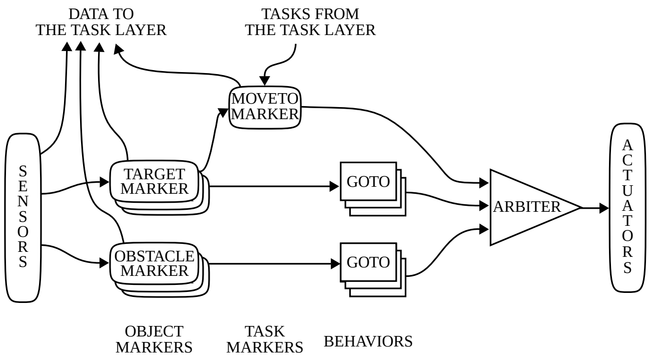


Fig. 16. A control system for reaching targets and avoiding obstacles. In addition to Goto maps, the behaviors produce Avoid maps.

The control system recognizes the two roles of objects, targets and obstacles. The Target and Obstacle markers are bound to the targets and obstacles. The definition of a target and an obstacle and hence the processing required to bind and update the object markers depends on the application. For example, when all moving objects are targets and all stationary objects obstacles, the binding and updating of the markers requires detecting the objects and classifying them as moving and stationary. This is done by the Sensors subsystem.

The control system contains one task marker: the Moveto marker describes a task to move near a target object. It forms the interface between the Samba and the task layer. The Moveto marker is connected to the Target markers. These markers represent the locations of the targets and propagate the data to the Moveto marker.

When no Moveto task has been given, the control system performs the predefined tasks coded in the arbiter. For example, the robot can follow the nearest moving target. When a Moveto task is given, the task overrules the predefined tasks. The activation signal of the Moveto marker sent by the task layer contains the location of the target to be reached. The Moveto marker stores this location. It selects from all the Target markers the marker nearest the given location and updates the location based on the selected marker. This operation continues as long as the Moveto marker is active.

The only difference between a predefined task and a task given by the task layer is how the target to be reached is determined. The arbiter performs both the predefined and the Moveto tasks by combining with the MAV method the Goto map of the selected target and the Avoid maps of the obstacles and other targets. Thus, the targets have also a more general role of being an obstacle. The arbiter attaches the role of a target to one object represented by target markers and the role of an obstacle to the rest of the objects represented by target markers. When there is one obstacle, the Moveto map might look like the maps in Fig. 15. The arbiter selects the action with the heaviest weight from the combined map and sends it to the actuators.

3.2 Robot actions

In this section, we describe the new representation for robot actions, i.e., the action map representation. We start by specifying the representation. We discuss the different types of action maps. In addition, we present methods for calculating action maps and for modifying and combining them.

3.2.1 The action map representation

Robot actions are represented as action maps. The action map representation is specified as follows:

> ***An action map*** *specifies for each possible action how preferable the action is from the perspective of the producer of the map. The preferences are shown by assigning a weight to each action.*

An action type is specified by the parameters of the action of that type. The most common action type is a velocity action, as the main task of the control system is to decide on the velocity of the robot at each moment. A velocity map is defined as follows:

> ***A velocity map*** *specifies for each possible velocity vector (direction, speed) how preferable the velocity is from the perspective of the producer of the map. The preferences are shown by assigning a weight to each velocity.*

A velocity map specifies weights for different velocity vectors (direction, speed). Velocity maps are three-dimensional: an action is described by a direction and a speed and for each action there is a weight. The weights form a surface in the polar coordinate system (direction, speed). Fig. 13 shows an example of a velocity map.

The control system modules reasoning robot motions produce velocity maps. Behaviors reasoning actions for different actuators produce different action maps. For example, a behavior reasoning command to kick a ball produces impulse maps. An impulse map is defined as follows:

> ***An impulse map*** *specifies for each possible impulse vector (direction, magnitude) how preferable the impulse is from the perspective of the producer of the map. The preferences are shown by assigning a weight to each impulse.*

The goal of the producer of an action map is to achieve a certain relation in the environment. In this work, the relation is most often a contact between two objects. One of the objects is *the controlled object*; either the robot itself or an object in the environment. Velocity maps are for controlling the robot, whereas impulse maps are for controlling an object in the environment. The other object in the relation, *the target object*, is not controlled by the robot. Hence, from an action map we can determine how a relation between a particular target object and the controlled object can be achieved by changing the state of the controlled object.

The producer of a map can also aim at some other relation than contact between the target object and the controlled object. For example, the goal of a producer might be to reach one meter's distance from the target object. Only contacts are considered in the following discussion. A map for achieving some other relation is in the simplest case calculated in a similar manner with a map for reaching a point, although some tasks may require the robot actions to be more constrained.

3.2.2 Calculating action maps

Action maps are calculated in two stages. First, the weights for *the satisfying actions* – those actions that result in a successful task execution – are calculated based on a global optimality criterion. The best action among the satisfying actions is called *the optimal action*. Second, the weights of the rest of the actions are calculated based on how much they give progress to the task (although they do not result in success). After this stage, all the actions giving progress to the task have a nonzero weight. These actions are called *the progressing actions*.

The global optimality criterion utilized in weight calculation is the time to collision. The shorter the time needed to reach the target object, the heavier the weight for that action. The relation between the time to collision and a weight is determined by a weight curve, such as the one shown in Fig. 17.

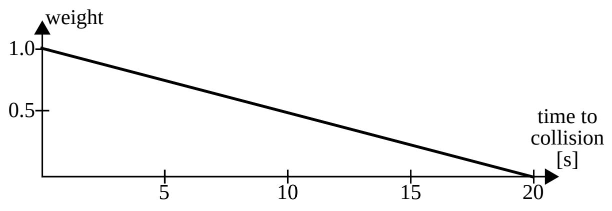


Fig. 17. A weight curve.

The primary reason for calculating weights also for the actions that do not result in a collision is to facilitate executing several tasks in parallel. A large amount of progressing actions increases the probability of some action belonging to the progressing set of each of the parallel tasks. That is, the larger the amount of alternative actions, the wider the variety of situations in which an action giving progress to each of the parallel tasks can be found.

We have developed two methods for map calculation, the Many Satisfying Actions method and the One Satisfying Action method. These methods are called hereafter the MSA method and the OSA method, respectively. All action maps presented in Section 3.1 are calculated with the MSA method. The methods differ in the criterion used to judge whether an action is satisfying. The satisfying actions are defined for these methods as follows:

In the MSA method, a satisfying action causes a collision with the target object. The number of satisfying actions is not limited.

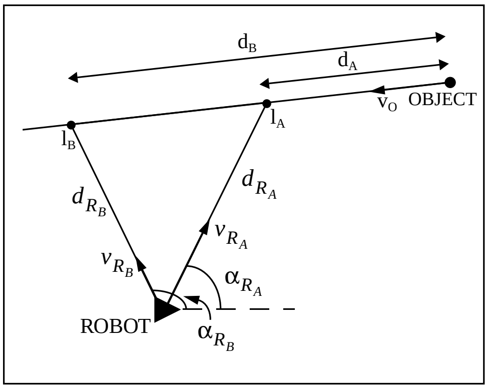
In the OSA method, only the optimal action is judged as satisfying.

3.2.3 The Many Satisfying Actions method

In the MSA method, all actions directing the robot towards a stationary object are satisfying. For a moving object, a satisfying action causes the robot to reach a point at the object trajectory at the same time with the object. The satisfying actions are calculated by shifting a point along the trajectory of the target and calculating at each location the time the target travels to the location. This is the time to collision. The coordinates of the location specify the robot heading required for a collision and the distance the robot has to travel. The speed of the robot required for a collision is calculated based on the distance and the time to collision. The heading and the speed specify a velocity action. The weight of the action is calculated based on the time to collision.

Fig. 18 illustrates the calculation of satisfying actions. The velocities causing a collision with a moving object are calculated for two locations, l_A and l_B. Also weights are calculated for the velocities. In the example, d_{R_n} denotes the distance from location l_n to the location of the robot. Further, α_n and s_n denote the direction and speed components of velocity v_n . The *weight* function maps a time into a weight as shown in Fig. 17.

The weights of the progressing actions are calculated by approximating the quality of each action based on the qualities of the neighboring actions in the map. This approach is based on the assumption that neighboring actions in a map produce nearly identical results. The quality of an action is approximated by propagating; by adjusting a weight towards the weights of the neighboring actions in the map. A propagating procedure goes through every action in an action map and calculates a new weight for each action based on its



The collision times:

$$t_A = d_A / s_O$$
$$t_B = d_B / s_O$$

The speeds and velocities required for collisions:

$$s_{R_A} = d_{R_A} / t_A \qquad v_{R_A} = (\alpha_{R_A}, s_{R_A})$$
$$s_{R_B} = d_{R_B} / t_B \qquad v_{R_B} = (\alpha_{R_B}, s_{R_B})$$

The weights for the velocities:

$$w_A = weight(t_A)$$
$$w_B = weight(t_B)$$

Fig. 18. Calculation of satisfying actions with the MSA method.

current weight and the weights of the neighboring actions. The structure of the MSA procedure is the following:

```
To calculate an action map using the MSA method:
• For each action resulting in a collision with the object,
    • Calculate a weight for the initial map based on the collision time.
• Call the initial map the current map.
• Until the map has been processed the given number of times,
    • For each weight in the current map, calculate a weight for a new map
      using the relaxation formula.
    • Make the new map the current map.
```

The relaxation formula for calculating a new weight for an action is as follows:

$$w^n_{i,j} = (1-p) \cdot w^{n-1}_{i,j} + p \cdot max(w^{n-1}_{i,j}, w^{n-1}_{i+1,j}, w^{n-1}_{i-1,j}, w^{n-1}_{i,j+1}, w^{n-1}_{i,j-1}) \qquad (3.1)$$

In Eq. (3.1), $w^n_{i,j}$ is the weight for the action (i, j) in the map. The i index specifies the direction of the action and the j index the magnitude. The propagating factor p defines how much the weight is adjusted towards the neighboring values. The parameter n defines the iteration number. One pass of the procedure produces weights for those actions that have a nonzero weight as an immediate neighbor. Hence, to produce nonzero weights for a wider set of actions, the procedure has to be executed several times.

Eq. (3.1) differs from the standard relaxation formula (Winston 1992, 244). To speed up the propagation, the weight is tuned towards the maximum of the neighboring weights instead of the average. Furthermore, to preserve the weights of the satisfying actions, weights are adjusted only upwards. This is accomplished by taking also the weight being adjusted into account when selecting the maximum weight. Fig. 19 shows an example of propagation.

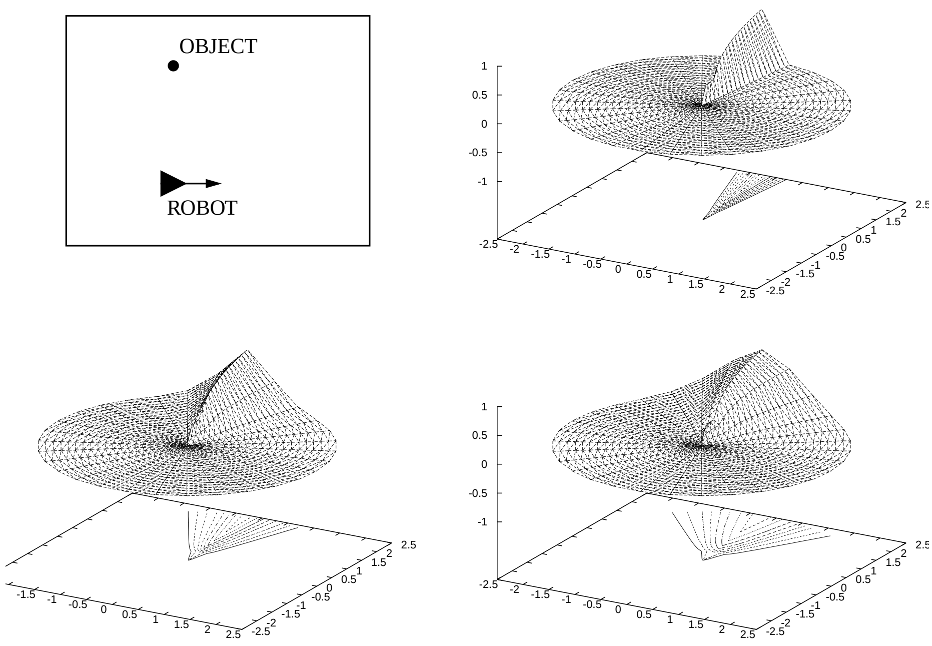


Fig. 19. An example of propagation. Top left: the situation. A stationary object is located at the left side of the robot. Top right: the weights for the actions reaching the object. Bottom left: the map after one propagation with a propagation factor of 0.15. Bottom right: the map after 8 propagations.

The action maps show in Fig. 19 are calculated for a point. The dimensions of an object are taken into account by tuning the propagation parameters. When the propagation factor and the number of propagation times are increased, the map contains heavy weights in a larger area. Another method for taking the dimensions into account is given in Section 3.2.4.

Propagation was selected as the first method for approximating the weights because it is easy to implement. Propagating has the disadvantage that it is difficult to ground the weights to the situation around the robot. The propagation parameters have to be tuned by hand in such a way that both primitive and composite maps reflect the situation around the robot. Especially guaranteeing that all possible combinations of primitive maps produce correct results is difficult. However, the experiments have shown that although tuning is difficult, propagating produces correct maps once the tuning is done.

This problem of weak grounding could be solved by extending the optimality criterion for all the actions in a map. This could be done by relating a weight to the shortest distance between the robot and the object that can be reached if the corresponding action is performed. In addition, the weight would be related to the time it takes to reach this distance. In this case, a weight surface would specify a weight for each (shortest distance, time to shortest distance) pair. Fig. 20 illustrates such a weight surface. The intersection of the weight surface and the plane spanned by the weight and time axes corresponds with the weight curve shown in Fig. 17.

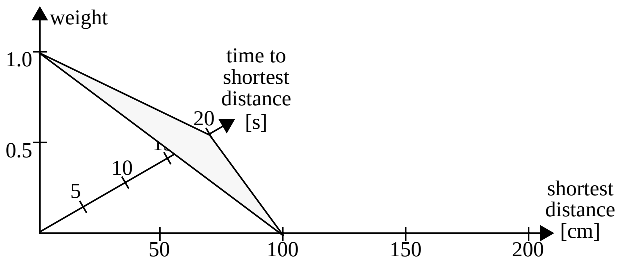


Fig. 20. A weight surface.

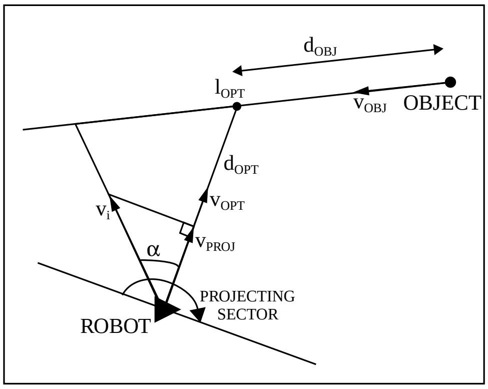
However, there is a simpler way to ground all weights. When only the optimal action is judged to be satisfying, the qualities of all the other actions can be approximated by projecting. This is the second method, the One Satisfying Action method, for calculating maps.

3.2.4 The One Satisfying Action method

When calculating an action map with the OSA method for a stationary object, only the action of driving towards the object at full speed is judged as satisfying. This action is the optimal one for reaching a stationary object. For a moving object, the optimal and hence the only satisfying action reaches the object at the earliest possible collision point. For the rest of the actions, the weights are calculated by evaluating how fast each action gives progress to the task of reaching the object at the optimal collision point. This is done by projecting the action into the optimal path and calculating the time it takes to travel the optimal path using the projected speed. The structure of the OSA procedure is the following:

To calculate an action map using the OSA method:
- *Calculate the optimal path and the optimal action.*
- *For each action in the projecting sector,*
  - *Project the action into the optimal path.*
  - *Calculate the time it takes to travel the optimal path by performing the projected action.*
  - *Calculate a weight for the action based on the calculated time.*

The projecting sector determines the progressing actions that get a nonzero weight by the OSA procedure. The center of the projecting sector is in the direction of the optimal action and the edges 90° to both sides of the optimal action. Fig. 21 illustrates the projecting approach for calculating velocity maps. In the figure, d_{OPT} denotes the distance from the location of the robot to the optimal collision point. Further, s_n denotes the speed component of velocity v_n . Fig. 22 shows an action map calculated by the OSA method.



The optimal collision time:

$$t_{OPT} = \frac{d_{OPT}}{s_{OPT}} = \frac{d_{OBJ}}{s_{OBJ}}$$

The projected speed:

$$s_{PROJ} = s_i \cdot \cos(\alpha)$$

The time to collision at the projected speed:

$$t_{PROJ} = \frac{d_{OPT}}{s_{PROJ}}$$

Fig. 21. The projecting method for calculating a weight for an action. The weight for the action v_i is calculated based on the time t_{PROJ} .

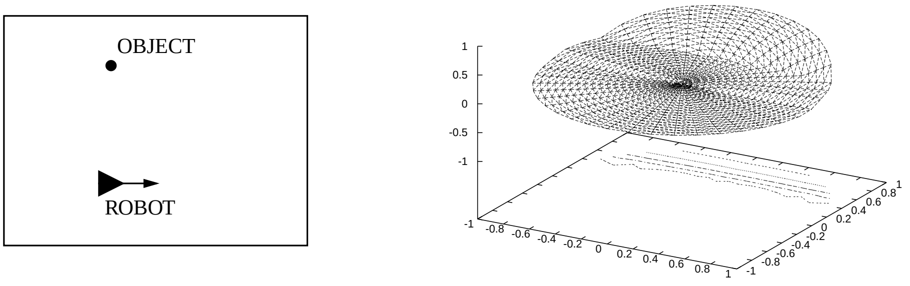
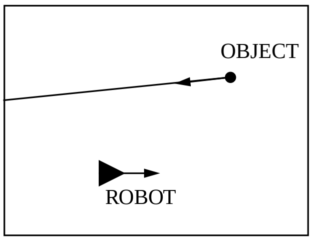


Fig. 22. A velocity map calculated by the projecting method. Left: the situation. Right: the map.

The velocity maps shown in Fig. 19 and Fig. 22 are calculated for a stationary object. Although in both the MSA and OSA methods the velocities in the direction of the object get the heaviest weights, the maps are quite different. In the MSA method, the weights decrease at the same rate in every direction, whereas in the OSA method the actions of the optimal speed are emphasized.

Fig. 23 shows action maps calculated by both methods for a moving object. The difference is significant. The MSA method produces actions for alternative collision points, for reaching the collision point at the same time with the object, whereas the OSA method produces actions for reaching the optimal collision point as soon as possible. That is, the MSA method produces actions for reaching the right location at the right time, whereas the OSA method produces actions for reaching the right location as soon as possible.



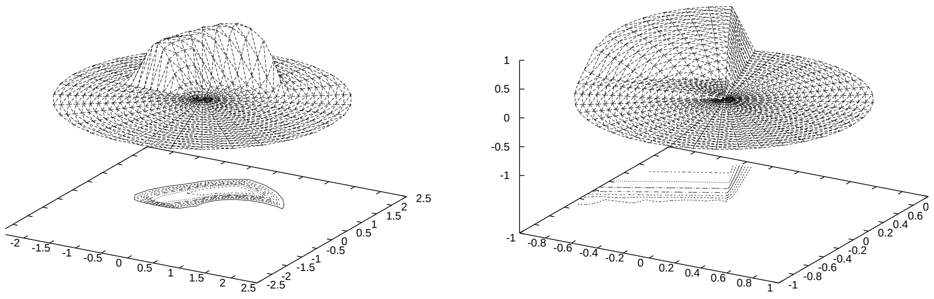


Fig. 23. Action maps calculated for a moving object by the MSA and OSA methods. Top: the situation. Bottom left: the MSA method. Bottom right: the OSA method.

When the map is calculated by projecting, there is one further point to be considered: the optimal action for reaching a moving object is to drive at the maximum speed towards the first possible collision point, as this is the action requiring the shortest time. The object cannot be reached if the robot turns more towards the object. As the object reaches the collision point in this direction in a shorter time than the collision point in the optimal direction, the robot should drive faster than the maximum speed. For this reason, the weights for the actions turning the robot over the optimal direction should have lower weights than the actions at the other side of the optimal direction. In the OSA method, these actions are given zero weights, as can be seen from Fig. 23.

The action maps above are velocity maps calculated for points; both the objects and the robot are supposed to be points. For objects with dimension, such as circular and rectangular objects, the procedures need to be modified. The size of the robot and an object need to be considered especially when the robot approaches the object. When the map is calculated by projecting, shape and size are taken into account as follows: the object is enlarged by the radius of the robot (the robot is assumed to be circular). For each action in the object sector, that is, producing movement towards the enlarged object, the weight is calculated based on the time it takes to reach with that action the edge of the enlarged object. This method of processing actions in the object sector can also be utilized in the MSA method.

For the actions in the left and right projecting sectors, the weights are calculated by projecting to the leftmost and rightmost direction pointing towards the enlarged object, respectively. The left projecting sector covers the directions from the leftmost object sector at a direction 90° anticlockwise, and the right projecting sector the directions from the rightmost object sector at a direction 90° clockwise. Hence, the structure of the OSA procedure for a circular object and robot is the following:

To calculate an action map using the OSA method for a circular object and robot:
- *Enlarge the object with the radius of the robot.*
- *Calculate the object sector.*
- *Calculate the optimal path and the optimal action.*
- *For each action in the object sector,*
  - *Calculate the time it takes to reach the edge of the enlarged object by performing the action.*
  - *Calculate a weight for the action based on the calculated time.*
- *For each action in the left and right projecting sectors,*
  - *Project the action into the nearest object sector direction.*
  - *Calculate the time it takes to reach the edge of the enlarged object by performing the projected action.*
  - *Calculate a weight for the action based on the calculated time.*

Fig. 24 shows velocity maps calculated for a circular object at different distances. The distances are measured between the center points of the robot and the object. As the distance to the object decreases, the sector covered by the object becomes broader. This is illustrated in Fig. 25. This approach of defining the projecting sector was selected because it produces wide weight peaks. Such peaks are useful especially in cluttered environments where obstacles can block a large proportion of the progressing actions.

3.2.5 Calculating impulse maps

Calculating impulse maps differs slightly from the calculation of the velocity maps. First, when calculating the impulse to be given to an object, the momentum of that object has to be taken into account. This makes the calculations more complex. Second, when calculating a velocity map for controlling a robot, velocities that do not achieve the goal but decrease the distance to it can be approved, as the velocity of the robot can be corrected later. For example, while circumventing an obstacle the velocity can differ considerably from the optimal one. When calculating an impulse map the constraints are stricter, as only one impulse can be given and that impulse should produce a momentum that causes the goal to be achieved. As a consequence, impulse maps cannot be propagated nor projected

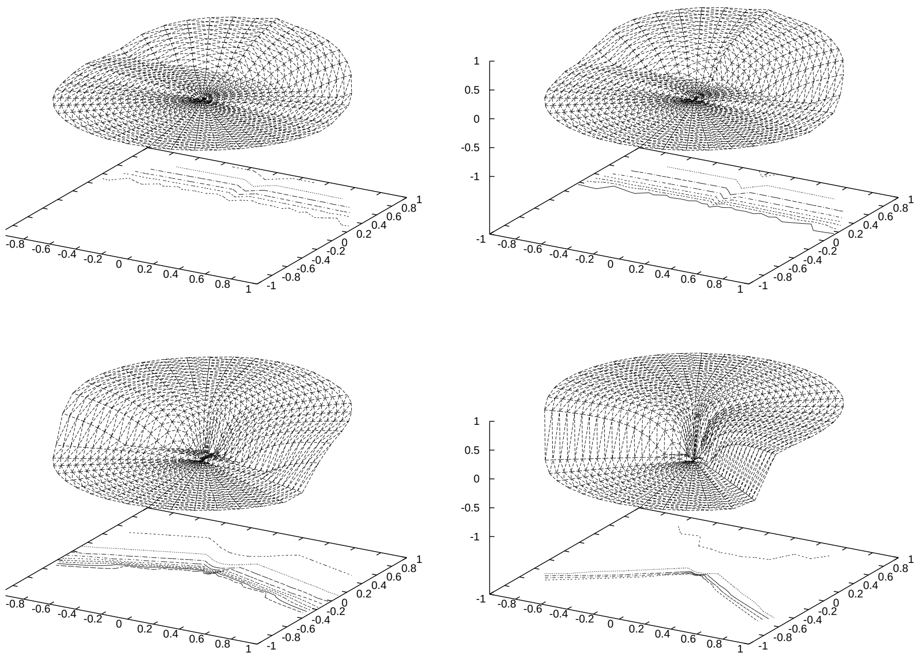


Fig. 24. Velocity maps for an object with shape. Both the object and the robot are circular and have a radius of 0.8 meters. Top left: the distance to the object is 15 meters. Top right: the distance to the object is 10 meters. Bottom left: the distance is 5 meters. Bottom right: the distance is 3 meters.

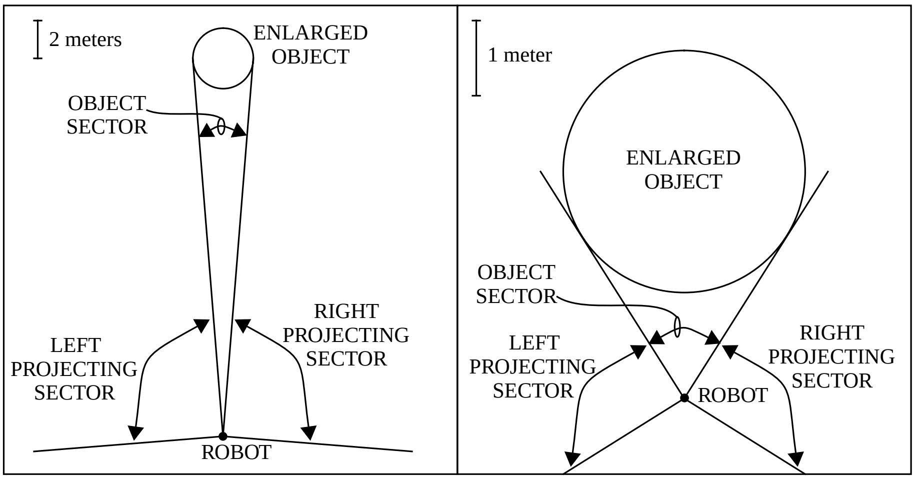


Fig. 25. The object and projecting sectors. Both the object and the robot are circular and have a radius of 0.8 meters. Left: the distance to the object is 20 meters. Right: the distance to the object is 3 meters

as much as the velocity maps. Otherwise the impulse maps can be calculated in a similar manner to the velocity maps.

As an example, Fig. 26 shows a situation in which the robot is playing soccer. The robot is to pass the ball to a moving team-mate. The actions with a heavy weight form a narrow ridge in the map, because the conditions for a successful task are strict: the ball has to be at the right location at the right time.

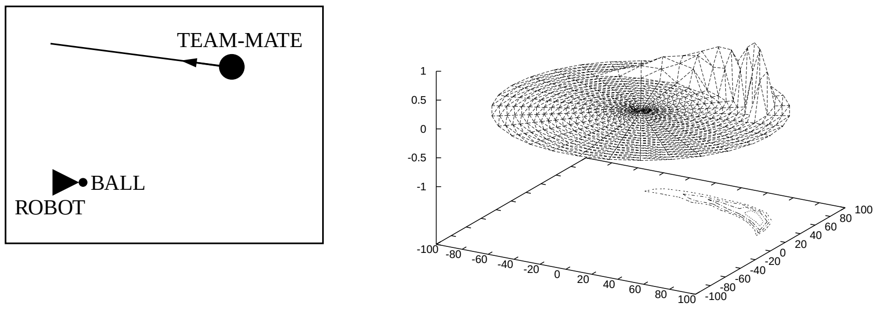


Fig. 26. An impulse map for passing a stationary ball to a moving team-mate. Left: the situation. Right: the map.

3.2.6 Goto and Avoid maps

All the discussion above is related to maps containing only positive weights. A map of this kind is called a Goto map, because performing the action with the heaviest weight on such a map causes the robot to reach the corresponding object. As a Goto map activates the robot, it is also called an *activating map*. The weight values of an activating map are in a range [0.0, 1.0]. *Inhibiting maps* contain negative weights. An inhibiting map is also called an Avoid map as it prevents collision with an obstacle when a target is being reached with the help of a Goto map. The weight values of an Avoid map are in a range [-1.0, 0.0]. Fig. 14 shows an Avoid map. An Avoid map can be produced either by negating the weights of the corresponding Goto map, or by a separate procedure. A separate procedure is required when negating a Goto map would cause too many actions to be blocked. Particularly, reaching a target located behind an obstacle can be a difficult task if the Avoid map for the obstacle is calculated by negating the Goto map.

3.2.7 Combining maps with the MAV method

The Goto and Avoid maps specified above are primitive, as they represent the preferred actions when a single object in the environment is considered. A composite map represents the preferred actions when several objects are considered. It is formed by modifying and combining primitive maps. Primitive action maps are modified by filtering operations and combined by the Maximum of Absolute Values (MAV) method, which is defined as follows:

To combine action maps with the MAV method, select for each action the weight with the greatest absolute value.

For each action, the shortest time it takes to reach an object is selected and the corresponding weight is stored on the composite map. Thus, the sign of a weight on a composite map specifies whether an obstacle or a target would be reached first if the corresponding action was performed. The absolute value of the weight specifies the time to collision with the object that would be reached first. The structure of a MAV procedure for combining two action maps is the following:

```
To combine two action maps using the MAV method:
• For each action,
    • Calculate the absolute values of the weights for the action.
    • Select the weight with the greater absolute value.
    • Store the selected weight with its original sign in the new map.
```

The general MAV procedure has the following structure:

```
To combine a set of action maps using the MAV method:
• Remove a map from the set and call it the composite map.
• For each map in the set,
    • Combine the map and the composite map with the MAV method
      for two maps.
    • Make the new map the composite map.
```

Fig. 15 illustrates the MAV method for compiling action maps. Fig. 27 shows a more detailed example. Actions from a Goto map and two Avoid maps are combined with the MAV method. The positive weights are from the Goto map and the negative ones from the Avoid maps. On the left, the absolute value of the Goto weight is greater than those of the Avoid weights, so the target would be reached with that action first. The Goto weight is stored in the composite map. On the right, an obstacle would be reached first, so a negative weight is stored in the composite map.

$$\begin{tikzcd}
\boxed{+0.76} \arrow[dr] & & & \boxed{+0.46} \arrow[dr] & & \\
\boxed{-0.62} \arrow[r] & \boxed{\text{MAV}} \arrow[r] & \boxed{+0.76} & \boxed{-0.62} \arrow[r] & \boxed{\text{MAV}} \arrow[r] & \boxed{-0.62} \\
\boxed{-0.28} \arrow[ur] & & & \boxed{-0.28} \arrow[ur] & &
\end{tikzcd}$$

Fig. 27. Combining actions from a Goto map and two Avoid maps with the MAV method. Left: the target would be reached first with the action being combined, so the positive weight is selected. Right: an obstacle would be reached first, so the negative weight is selected.

3.2.8 Filtering

Sometimes the conditions set for a task need to be modified. For example, only actions that reach the target fast enough might be allowed. Or, when avoiding an obstacle, those actions that cause a collision only after a considerable length of time might be allowed. Modifications of this kind can be produced by filtering. Filtering is defined as follows:

To filter an action map, set the weights of the actions fulfilling the filtering condition to zero.

The filtering condition specifies the actions to be removed from the map. It can be, for example, a lower limit for the time it takes to reach the object. The filtering condition can also constrain directly an action parameter, for example the speed of the robot. Fig. 28 shows the result of filtering actions that require too long a time to reach a target.

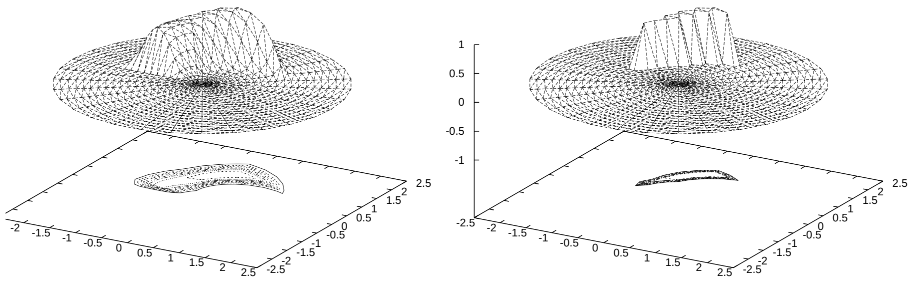


Fig. 28. Effect of filtering. Left: Goto map. Right: the actions requiring more than 1.1 seconds to reach a target have been filtered from the Goto map shown on the left.

Filtering can also be performed by utilizing a filter mask containing a zero for each action to be filtered and one for the rest of the actions. Filter masks are convenient especially when the same filtering is to be performed on several maps, as the actions to be filtered only need to be calculated once. Furthermore, the actual filtering can be performed very efficiently by utilizing the filter mask.

A situation in which a stationary target is located behind a stationary obstacle illustrates the effect of filtering. A Goto map is calculated for the target and an Avoid map for the obstacle. Without filtering, the Avoid map has heavy weights for all the actions in the direction of the obstacle. As the obstacle is nearer than the target, all the actions in the direction of the obstacle (and the target) are forbidden in the map compiled from the Avoid and Goto maps. This situation is illustrated in Fig. 29.

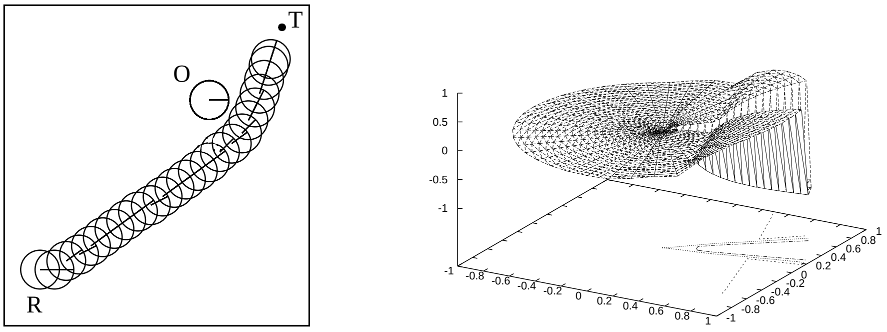


Fig. 29. An example of an obstacle (O) preventing the actions of the robot (R) in the direction of a target (T). Left: the trace. Right: the map compiled from the Avoid and Goto maps after the robot has turned. As the obstacle is nearer than the target, it prevents all actions of driving towards the target until the robot has circumvented the obstacle.

But preventing all directions pointing towards an obstacle is not desirable when the obstacle is not near. In an environment populated by many obstacles, this can result in preventing all progressing actions in the Goto map. This problem can be solved by filtering; by removing from the Avoid map all actions requiring more than, for example, 0.5 seconds for a collision. As a result, as long as the robot is far enough from an obstacle, all the actions in the corresponding Avoid map are filtered and the robot is allowed to move towards the obstacle. When the time to reach the obstacle with an action goes below the threshold (in this case, 0.5 seconds), the Avoid map prevents that action. Obstacle avoidance with filtering is illustrated in Fig. 30. The robot drives first towards the obstacle and starts to circumvent it when the actions causing a collision with the obstacle would take less than 0.5 seconds to complete.

In addition to modifying the obstacle avoidance behavior, filtering can be utilized in achieving goals in parallel. An action map calculated on the basis of the constraints set by a goal can be filtered based on the constraints set by another goal. For example, a robot can have a goal to drive towards a location and a goal to reach a moving object. The constraints set by the goal of reaching the moving object can be encoded in an action map. The

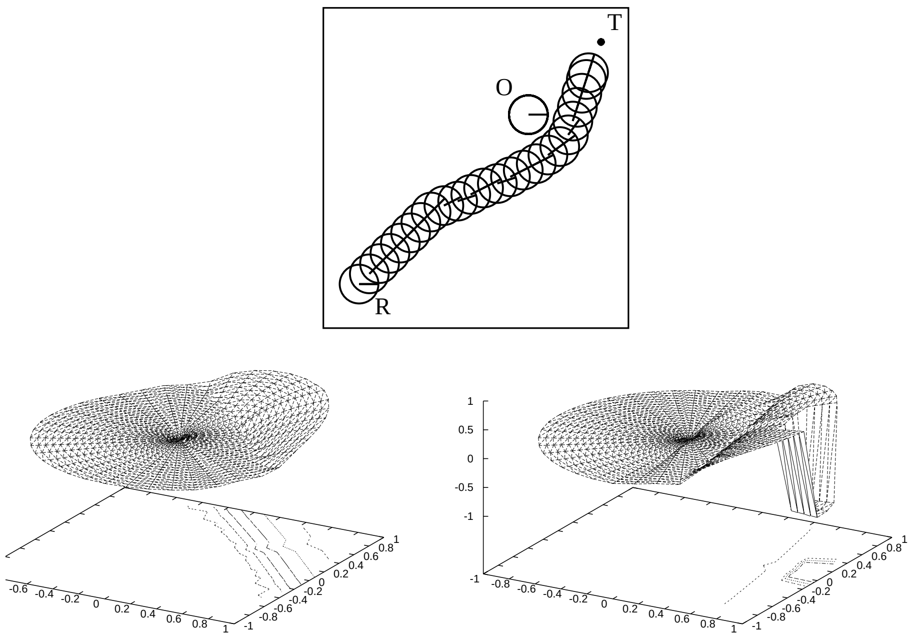


Fig. 30. An example of the robot (R) driving towards a target (T) although an obstacle (O) is located in front of the target. Top: the trace. Bottom left: map compiled from the Avoid and Goto maps at the initial situation. The situation is the same as in Fig. 29 but this time all actions requiring more than 0.5 seconds for a collision are filtered from the Avoid map. No actions are forbidden and the robot drives towards the obstacle. Bottom right: map compiled from the Avoid and Goto maps when the robot approaches the obstacle. The fastest actions in the direction of the obstacle are forbidden and the robot starts to circumvent the obstacle.

constraints set by the goal of driving towards the location can be taken into account by filtering the action map. The filtering condition specifies how much the robot is allowed to deviate from the path leading to the goal. The condition defines a sector. After filtering, only the actions for reaching the moving object at this sector have a nonzero weight. This example is illustrated in Fig. 31. When the Goto map calculated for the moving object is not filtered, the robot reaches the object at the shortest possible time. When the map is filtered based on the other goal of driving towards the location, the robot selects an action giving progress to both of the tasks. As a result, the robot reaches the moving object while driving towards the goal location. The maps in Fig. 31 are calculated using the MSA method.

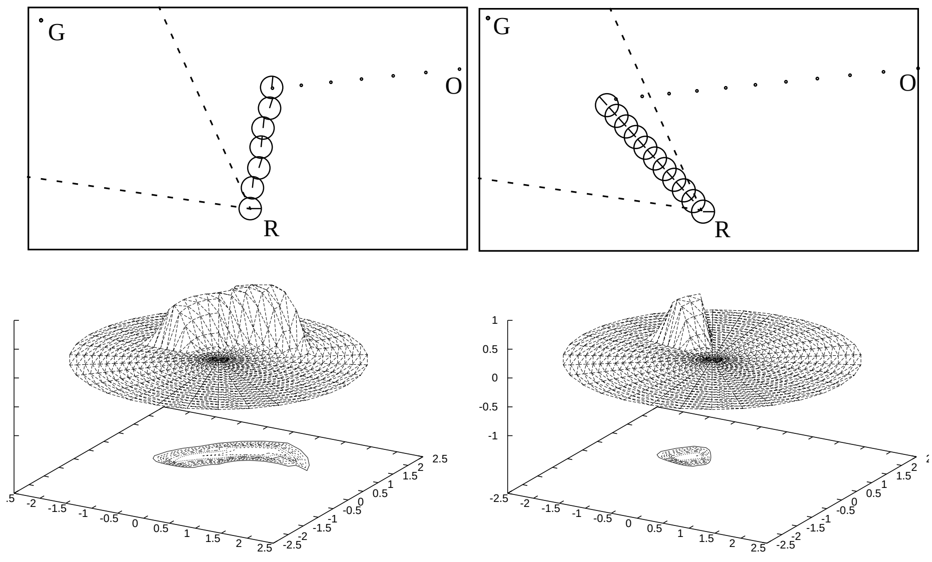


Fig. 31. An example of the robot (R) driving towards a goal location (G) and reaching a moving object (O) at the same time. Top left: the trace with no filtering. The robot reaches the object at the optimal location but does not drive towards the goal location. The rays mark the allowed deviation from the path leading to the goal. Top right: the trace with filtering. The robot reaches the object and drives towards the goal location. Bottom left: the Goto map calculated at the initial situation for the moving object without filtering. Bottom right: the resulting map when the actions causing too large a deviation from the path are filtered from the Goto map.

3.3 Markers

Markers are modules describing intermediate results and goals. The Samba architecture contains three types of markers: object, task and attention markers. Object markers are bound to the most important objects by the sensor system. Task markers describe task-related information. They are bound to the objects in the environment by the task layer. Attention markers describe the attention of the behaviors. The following sections describe each marker type in more detail.

3.3.1 Object markers

The robot represents the state of the environment by the object markers. They connect to behaviors the objects that the robot is going to manipulate or should otherwise consider when choosing the next actions. Object markers store data about the objects, update the data automatically based on sensor data, and communicate the data to the other modules. They are modules storing intermediate results.

An object marker represents all the data about an object the other modules need in their calculations: the location, orientation, velocity, and other observable features of an object. For example, an object marker describes the size and shape of large objects, as this data is needed in producing action maps. The location of an object is the most important piece of information about an object, as the location is used to index the object.

Objects are classified according to their roles, as was discussed in Section 2.3. A role set is attached to each object marker. The identity of the object marker specifies the most specific role of the object represented by the marker. The object markers are connected to other modules based on the roles. The number of object markers depends on the application; how many roles and objects of a role the robot needs to take into account.

Object markers are activated (bound) and updated by the sensor system. The markers in the egocentric coordinate system update the location of the object also on the basis of the robot's ego-motion. The markers in the image coordinate system are updated based only on observations.

The processing required to bind object markers to objects in the environment and to update the markers depends on the application. The sensor system might bind all object markers automatically according to the situation around the robot, or consider also the task markers when selecting the objects the robot needs to take into account. The sensor system might be capable of updating in parallel all object markers or only a part of them. Behaviors might also participate in the binding and updating by directing the sensors. In this work, no general method is defined for binding and updating object markers. Instead, a specific mechanism is defined for each application separately.

3.3.2 Task markers

Task markers form an interface between the Samba control system and the task layer which reasons the tasks to be executed. They describe goals for the Samba system. When the task layer wants a task to be executed, it activates the corresponding task marker. The activation can be interpreted as binding task parameters to an object in the environment. This causes the Samba system to produce actions for manipulating the object or for reacting otherwise to it. In other words, task markers connect task-related environment objects to actions.

Task markers attach different roles to environment objects: for one task marker an object might be an object to be transferred, for another the same object can be a target to be tracked, or an obstacle to be avoided. Hence, several task markers can index the same object.

A task marker contains all the data that is needed by the modules of the Samba control system to perform the task. However, no complete data about the task is needed. It suffices to define how the default operation of the system needs to be changed. For example, when a particular object is to be reached, there is no need to specify that all the other objects are to be avoided.

The information encoded in a task marker specifies a goal for the robot. A location is the most important piece of information about a goal. The location is related to an object; it is either the location of the object, or has a particular relation to the object. In a similar manner to object markers, goals are indexed by their locations. In addition, a task marker represents the operations the robot should perform at the location. The operations need not be represented explicitly but may be identified according to the identity of the marker. Task markers can contain also time constraints. These constraints specify, when the goal should be reached. Examples of such constraints are “in two seconds”, “exactly five seconds from now”, and “not before four seconds”.

A task marker sends signals to the arbiter. The arbiter performs the task by modifying and combining the action maps as required by the task. When the execution of a task requires a map achieving some other relation than a contact, the corresponding task marker sends signals also to the behavior producing the map. In this case, the arbiter modifies and combines this map and the other maps required by the task.

Once the task layer has activated a task marker, the location stored in it need to be updated as the robot moves and the environment changes. As the task layer has the object markers available, it can continuously update the location and send it to the task marker. However, this method can be slow, depending on the amount of computation performed by the task layer. On the other hand, a task marker can itself continuously update the location on the basis of the data received from the object markers. This method is faster than the first one, but it requires a more complex task marker and more connections between the modules. Hence, when selecting the updating method, a compromise is needed between the delay of updating and the complexity of the control system.

In this work, the updating method is selected by comparing the complexities of the whole system and the delays the alternative methods produce. As the task layer calculates the values of the task marker when activating it, the whole system is simpler when the task layer also updates the values. When this update method is too slow, the task marker updates the values itself based on the object markers.

3.3.3 Attention markers

Attention markers are a special type of a marker. The markers are not necessarily bound to objects in the environment and they need not necessarily trigger task execution. Instead, they describe the past, current, and future attention of the behaviors and the sensors. Attention markers have similar advantages to these markers bound to objects: the markers can be used to communicate the behaviors' attention to other behaviors and to arbitrate the attention. Attention markers resemble task markers. The updating mechanism is different, however. Some attention markers update the location by sensor data, some by behavior commands, and some by the same mechanism as task markers.

For example, when the robot is equipped with active stereo vision, the fixation point of the cameras can be described by an attention marker called a fixation marker. This marker describes the externally observable focus of attention of the robot. Further, it connects the egocentric and image coordinate systems. The origins of both the image coordinate systems (left and right) are located at the fixation point in the egocentric coordinate system. The fixation marker updates the location based on sensor data describing the camera angles. Another example is an attention marker describing the area to be searched by sensors. In this case, the marker is updated by a behavior.

3.3.4 Coordinate systems

Markers can be described in several different coordinate systems. For example, the goals of the robot are represented in the egocentric robot coordinate system and the objects perceived by a vision system are represented in an image coordinate system. A location can be described in an image coordinate system as a point (x-coordinate, y-coordinate). In the egocentric coordinate system, a location is described as a vector (direction, distance). Markers in different coordinate systems need to be associated with each other in order to exchange data between them. For example, to get some extra information about an object described by a marker in the egocentric coordinate system, we need to know the corresponding marker in the image coordinate system.

The markers in the egocentric and image coordinate systems are associated in the same manner as the markers are associated with the objects. Each marker in the egocentric coordinate system indexes the corresponding marker in the image coordinate system by its location (i.e. image coordinates). The location of an egocentric marker in the image is calculated by the sensor module that updates the location of the marker. As the sensor module uses image data to update the egocentric marker, image coordinates are a by-product of the calculation. For example, the Zdf sensor module calculates the image region the cameras are fixed on (Zdf stands for Zero disparity filter, see Chapter 4). When there is an object at the intersection of the optical axes of the cameras, a Zdfcent marker is created at image coordinates (0,0). The corresponding location in the local environment is calculated based on camera angles and an egocentric Fix marker is bound to that point. The nearest object marker is bound to the same point (if near enough). The image coordinates of the both markers are set to (0,0).

An egocentric marker corresponding to an image marker can be found in a similar manner. The sensor module updating the egocentric marker based on the image marker stores the coordinates of the egocentric marker in the image marker. Hence, there is no special mechanism for linking different coordinate systems, but the sensor modules transforming data between coordinate systems store in a marker locations in both coordinate systems.

3.4 Behaviors

A behavior controls an actuator. It transforms signals from sensors and markers into commands for the actuator. Behaviors are the modules that get things done; each behavior knows how to solve some task. A behavior has the structure of the basic module depicted in Fig. 2. The signals received from sensors and markers are written in the input buffers and the output signals calculated by the behavior are written in the output buffers.

Behaviors send commands to the actuators they control continuously as long as the task is under execution. By sending commands they reserve the actuator for themselves and prevent other behaviors from controlling it. This approach produces a system in which many behaviors are active and the arbiters choose which commands to pass through.

Behaviors are closely related to object markers. For each object marker, there is a behavior receiving data from the object marker. The behavior produces primitive reactions such as “go to the object” and “avoid the object”. These reactions are in the form of primitive action maps. A behavior produces two different action maps: a Goto map and an Avoid map. Some behaviors produce action maps based on task markers, as was discussed in Section 3.3.2. Behaviors can also produce commands in some other form, for example, as velocity vectors.

3.5 Arbiters

An arbiter receives a set of commands, selects a subset of the commands, combines them, and sends the combined command to the actuators. In addition to the commands, an arbiter can also use sensor and marker data in selecting the most important commands. The number of arbiters is specified on the basis of the available actuators and how they are used. If only one actuator is active at any given time, one arbiter is sufficient. On the other hand, if several actuators are controlled continuously, each of them might be eligible to have their own arbiter. In this case, a mechanism might be required for coordinating the actuators.

An arbiter operates in three stages. First, the arbiter selects the tasks to be executed. The alternatives are the predefined tasks and the tasks given by the task layer via task markers. The arbiter utilizes the method of dynamic priorities in this selection. If a single task is selected and the command causing progress in the task represents a single set of command parameters, the actuator sends the command to an actuator. Otherwise the commands causing progress in the tasks are represented as action maps and the actuator proceeds to the second stage. At the second stage, the arbiter modifies the action maps and combines the modified maps with the MAV method. At the third stage, the arbiter selects the best action from the combined action map and sends it to an actuator.

The arbiter takes time constraints into account at the second stage. The time constraints can be either given a priori or defined by a task marker. For example, an object might be reachable between two and 15 seconds from now and the time constraint could be “in five seconds”. The arbiter filters from the map the actions that do not fulfill the time constraint.

The arbiter might utilize action maps also at the first stage when deciding the most important task. This approach has the advantage that the dynamics of the environment can be taken into account. That is, if a distance to a target is used as a condition for reaching the target, the motion of the target while the robot approaches it is not taken into account. The shortest time to reach the target, calculated on the basis of the best action in the corresponding action map, gives a more accurate approximation of the importance of the task and of the chances to reach the target.

The predefined tasks form the basic behavior that the robot produces when no tasks are given by the task layer. Although some simple tasks can be executed without action maps, most of the predefined tasks are defined as operations on action maps. In each task, the target(s) to be reached and the obstacles to be avoided are selected first. Then, a composite map is formed by combining a Goto map for each target and an Avoid map for each obstacle. Filtering operations can be used both for the primitive maps and the composite map for constraining the set of alternative actions. The maximum weight in a composite map denotes the optimal action from the perspective of the task. The tasks given by the task layer are performed in a similar fashion to the predefined tasks.

3.6 Discussion

The important features of the Samba control architecture are that it is reactive, goal-oriented, capable of executing tasks in parallel, and capable of understanding environment dynamics. Furthermore, it offers a variety of tools for decomposing a control system into subsystems. These features are achieved by extending behavior-based architecture with the representations of goals, intermediate results, and robot actions. Further, marker and arbiter modules are added in the architecture. Task and object markers describe goals and intermediate results. Action maps describe robot actions. Action maps are produced by behavior modules and combined by arbiter modules. Both types of modules utilize marker data in their calculations. The new modules and representations do not destroy the reactivity, because they do not extend the signal paths, increase the amount of computation, nor the amount of stored information considerably. In other words, as all actions are reasoned from the current environment state and at the pace of the environment, the Samba architecture is situated.

Task markers form an interface to a task layer which reasons the tasks to be executed. A Samba control system forms a hybrid system together with the task layer. The advantages of such a system were discussed in Chapter 1. Tasks are executed by modifying and combining primitive reactions represented as action maps. Actuator commands are not calculated based on the task constraints. Instead, behaviors calculate primitive action maps for the important objects in the environment. These maps are then modified and combined by the arbiter according to the task constraints. That is, the task layer does not calculate robot actions directly but guides the arbitration of the actions produced by the reactive layer. The rationale behind this approach is that as all actions of a robot are reactions to objects in the environment, all the actions needed to execute a task can be produced by modifying the primitive reactions. This is a major factor in achieving situatedness, as robot actions are always reactions to the latest sensor data, even if the actions were planned at the task layer. The importance of the deliberative system guiding instead of controlling directly the robot actions has also been stressed by Gat (1993).

A consequence of this approach, of executing tasks by modifying and combining the reactions on objects in the environment, is that all the actions of the robot are reactions to observable features. We have found this a reasonable assumption. As the main task of the robot is to interact with the environment, it is surrounded by observable features all the time. We admit that some situations require goal locations that are not observable. In soccer, for example, each player can have a default location specified on the basis of its role. As there are not many stationary objects in the field, a player cannot be instructed next to an observable feature. A similar situation arises when a robot is cleaning a large hall. However, even in these situations the goal location can be associated with observable features. The goal location can be represented in a local coordinate system spanned by observable features. That is, the goal location has a particular relation to the features. For a robot cleaning a hall, the coordinate system can be spanned by landmarks on the wall. For a soccer player, the coordinate system can be spanned by lines and flags. In the Samba architecture, task markers can represent goal locations that have a particular relation to observable features. This case was discussed in Section 3.3.2.

As pointed out in the first two chapters, behavior-based architecture suffers from the lack of representations. This problem hinders implementing the behavior coordination methods required by a goal-oriented operation. The Samba architecture contains two representations addressing this problem: markers and action maps. Task markers describe goals and serve as an interface between the reactive Samba system and the task layer. When a task marker is activated, it activates all the behaviors participating in the task execution and sends the task parameters to the behaviors. Task markers can also be used to deactivate the behaviors that prevent the goal from being achieved.

The action map representation of robot actions enables such behavior coordination that several behaviors are satisfied in parallel. In other words, the action map representation enables parallel task execution. The key point of action maps is that they preserve information. A primitive action map determines all the possible different actions causing progress in a task. The map combining method utilized by arbiters, Maximum of Absolute Values, preserves this information. As a result, those actions having heavy weights in a composite action map help several primitive tasks to progress in parallel. That is, the actions cause the robot to be at the right location at the right time from the perspective of several different primitive tasks. These actions do not necessarily give maximum satisfaction to any one primitive behavior, but satisfy several primitive behaviors partially. Specifically, action maps allow tasks to be assessed separately and combined by a simple operation. This advantage facilitates incremental development.

In this work, two methods were suggested for calculating action maps, the MSA method and the OSA method. The OSA method produces action maps for reaching a target as soon as possible. When the target does not need to be achieved in the shortest possible time the MSA method produces better results, particularly for a moving target. This is because the MSA method produces actions with the slowest sufficient speed for reaching a target whereas the OSA method favors the maximum speed.

The work on the action maps has not yet been completed. The MSA method could be improved to ground all weights in the environment. This could be done by extending the optimality criterion for all the actions in the map, as was discussed in Section 3.2.3. Furthermore, new procedures could be developed for calculating action maps for objects found from indoor environments. The filtering methods could also be extended. With more complex filtering conditions, a map for reaching a moving target could be filtered to also satisfy the goal of reaching the target inside a specified region. Finally, a new map combination method could be implemented for reaching several targets in parallel. This method would store in the combined map a nonzero weight only for actions that reach the targets at the same time. For those actions the weight in each map being combined is equal.

When utilizing action maps, it is possible to produce a composite map with no positive weights. This situation resembles the case of local minima in the potential field approaches. In such a situation, objects to be avoided prevent all actions reaching a target. To escape such a situation, a robot can either try to produce the same composite map with differently filtered primitive maps, or try another composite map. When the last composite map to be tried produces actions in all reasonable situations, the robot can escape from these situations.

Action maps allow time provisions. Rules such as “Go after the moving target if you can reach it in less than two seconds” are easily implemented by operations on action maps. Furthermore, action maps facilitate taking the environment dynamics into account. A primitive action map contains high weights for actions that cause an object to be reached, even if the object is moving. By combining several such maps an arbiter can, for example, avoid a moving obstacle while reaching a moving target. Further, the arbiter calculates the actions in a similar way for both static and dynamic objects as the information about how to take the dynamics into account is coded into action maps. An arbiter needs no extra reasoning to react to a moving object. As a result, it is possible to meet primitive goals in parallel in a dynamic environment. This is a central difference between this and the related work. This difference is caused by the amount of information encoded in a command. A larger amount of information encoded in action maps allows more versatile command combination operations than the command representations in the related work.

In the related work, information is lost on the signal path and hence executing tasks in parallel is difficult. Even if an individual module considers alternative actions for satisfying a primitive goal, these alternatives are not sent further. As a result, the commands produced by the modules do not contain enough information for a combination operation that would satisfy all modules. That is, the representation of robot commands prevents such combination operations. Although in some representations some alternative commands are represented and thus more versatile combination operations are possible, no representation allows a combination operation that would satisfy in parallel goals that require environment dynamics to be considered. As in a dynamic environment, the robot has to reach the right location at the right time, only producing alternative velocity commands allows selecting commands that satisfy several goals in parallel.

Alternative commands are sent further in the DAMN architecture (Rosenblatt & Thorpe 1995). The action map representation differs from the representation used in the DAMN architecture in two key points. First, in the DAMN architecture, the speed and direction commands are produced separately, whereas in the Samba architecture a single command contains both the speed and the direction. Second, in the Samba architecture the weights are grounded in the environment, as a weight determines the time it takes to reach a goal with the corresponding action. In the DAMN architecture, votes have no such interpretation. Actually, Rosenblatt states that “the semantics of the vote values are not well defined” (Rosenblatt 1997, 50). As a result, the methods for producing robot actions and combining them are quite different in the Samba and DAMN architectures.

An arbiter in Samba is more complex than the arbiters (or corresponding modules) in the related work. This is because action maps are a more complex representation that enables more complex operations for selecting a subset of the suggested actions. First, the modification (filtering) operations constrain the set of actions an action map suggests. For example, demanding that the target must be reached in less than two seconds inhibits a subset of the actions suggested by an action map by setting the weights of this subset to zero. Second, the MAV combining operation constrains the set of actions by demanding that an action must cause a target to be reached before an obstacle. Third, selecting the action with the heaviest weight produces the best action for progress of the task at hand. Furthermore, in the Samba architecture, the priorities of the behaviors are dynamic. They are calculated on the basis of the environment state and the task being executed.

The lack of representations makes it difficult to manage system complexity. The method for managing the complexity is to decompose the system into independent subsystems. Common representations are needed at the interfaces of these subsystems. The action map representation of robot actions and the representations of goals and intermediate results utilized by markers are such representations. Markers and action maps facilitate system decomposition in several ways. Task markers serve as a horizontal interface between the reactive and the task layer. Object markers serve as interfaces between data producers and consumers. Both horizontal and vertical interfaces hide implementation details of sets of modules from the rest of the system. These sets of modules are accessed through the markers.

Object markers facilitate managing complexity also by reducing the amount of computation. The behaviors receive from these markers the location and velocity of the objects, so they do not have to compute the locations and velocities from raw sensor data. Decomposing a control system into producers and consumers of sensor data has been utilized by many other researchers. However, we propose also decomposing the control system into producers and consumers of primitive reactions. Primitive reactions for all important objects in the local environment of the robot are produced by an independent, general subsystem. Primitive action maps form an interface between the producers and consumers of primitive reactions. Higher layers execute tasks by combining and modifying these primitive reactions based on task constraints.

The main difference in the use of markers between the Samba and Chapman’s Sonja architecture (1991) is that in the Samba architecture, markers form an interface also between a layer reasoning tasks and a layer performing them. In both systems, markers serve as an interface between data producers and data consumers. The comparison between the deictic representations (Agre & Chapman 1995) and the object and goal modules of the general behavior-based architecture applies directly to object and task markers. Both the deictic representations and the markers in the Samba architecture serve the same purpose of constraining the amount of things to be represented, but the deictic representations classify things more accurately relative to the circumstances and purposes of the robot. Object markers resemble more the markers of the Sonja system than the aspects. In Samba, aspects are reasoned according to object markers. Further, object markers do not describe the purposes of the robot. Task markers resemble more the aspects, as they describe the purposes of the robot but neither describes the circumstances directly.

4. Pure: experiments on markers

In this chapter, we describe the first set of experiments that was run at the Electrotechnical Laboratory, in Japan. In these experiments, we tested the first version of the Samba architecture, called Pure, that contained markers but no action maps. We used a mobile robot equipped with active vision. The second set of experiments is presented in the fifth chapter. In both sets of experiments, the primary goal was to show how the extensions for a behavior-based architecture, developed in this work, facilitate fulfilling the requirements for reactive task execution. Furthermore, the aim was to prove that these extensions do not destroy the reactiveness of a behavior-based system. A secondary goal was to illustrate the properties of the control architecture.

The architecture for Pose, Unblock, and Receive (Pure) controls a mobile robot to help other robots in their transfer tasks. The robot follows other robots (Pose). If some other robot is blocked by an obstacle, the robot pushes the obstacle away (Unblock). If some other robot releases its cargo, the robot takes over the transfer task (Receive). The robot is equipped with a stereo gaze platform that has 2 DOF vergence control (see Fig. 32). The robot has one onboard processor for controlling the mobile base and another for the gaze platform. Images are processed on a Datacube system and the Pure control system is executed on a host computer.

4.1 Architecture

The Pure architecture is shown in Fig. 33. Buffers are networks of markers. Task behaviors, task arbiters, and task markers in the Mobile Space Buffer form the task layer producing tasks for the robot and gaze behaviors. Robot behaviors control the mobile base and gaze behaviors the stereo gaze platform. Robot and Gaze are actuator modules. Sensors & Preprocessing includes all sensor modules (see Table 2). The IR sensors report distances to obstacles in front of the robot. The SelfMotion sensor detects the motion of the robot and the CameraAngles sensor the directions of the two cameras. The Vision sensor produces images that are processed by the ZDF and OpticalFlow sensors. The ZDF sensor implements the Zero Disparity Filter. It extracts from the image data the features that are at the center of both cameras, that is, the features that the cameras are fixated on

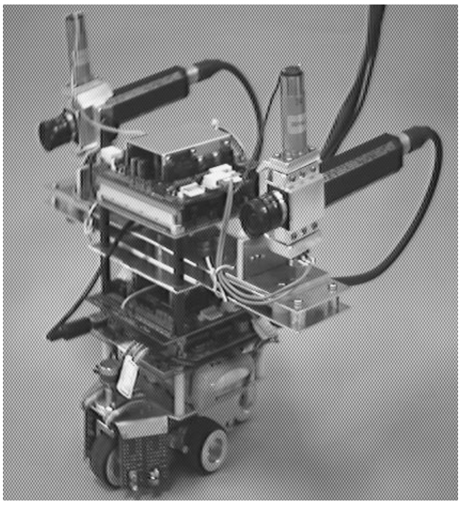


Fig. 32. The robot.

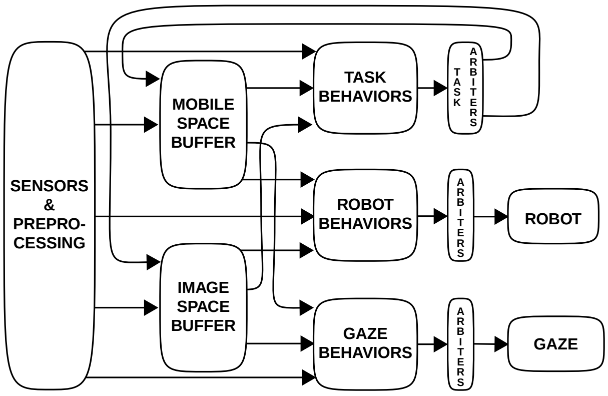


Fig. 33. Pure architecture.

Table 2. Pure sensors.

IR	SelfMotion	CameraAngles
Vision	ZDF	OpticalFlow
Predict	MarkHome	FitTrajectory

(see Fig. 38). The ZDF and OpticalFlow sensors are described in more detail in (Rougeaux *et al.* 1994, Kuniyoshi *et al.* 1994). The Predict, MarkHome, and FitTrajectory sensors are described below. They are virtual sensors.

The Mobile space buffer (MSB) contains markers in a local, robot-centered coordinate system. The features described by the mobile space markers are points in the environment and relations between them. The Image space buffer (ISB) contains markers in the image coordinate system. Table 3 shows the markers and their types.

Table 3. Pure markers.

Buffer	Marker	Marker Type
MSB	Fix	attention, object
MSB	Moveto	task
MSB	Transfer	task
MSB	Trajectory	object
MSB	Sarea	attention
ISB	Zdfcent	object
ISB	Home	object
ISB	Robot	object
ISB	Obstacle	object

The Fix (fixation point) marker describes the point the cameras are fixated on and also if there is an object at that point. Hence, it is both an attention and object marker. The fixation point is calculated based on the camera angles. The existence of an object is determined by the ZDF sensor. The Fix marker is the only mobile space marker that is active all the time. All the other mobile space markers are activated by behaviors and sensors. The Fix marker sends data to all the other mobile space markers. The markers update their feature values when the distance between the fixation point and the feature is small enough. The markers update their data also based on robot self motion. The markers use the self motion only if the Fix marker is too far from them or it is not on an object.

The Moveto marker contains pose and reference points. The reference point is associated with an object. The pose point is a goal position for the robot. The marker updates the reference point based on the Fix marker and the pose point based on the reference point and the relation between the points (see Fig. 34). This relation is specified by the behavior initializing the marker. Updating the Moveto marker is an example of how a task marker is updated by an object marker. However, this is a simple example, as the Moveto marker needs not select from among object markers.

The Transfer marker contains approach, object, and goal points. The object point is associated with an object. The approach and goal points have predefined relations with the

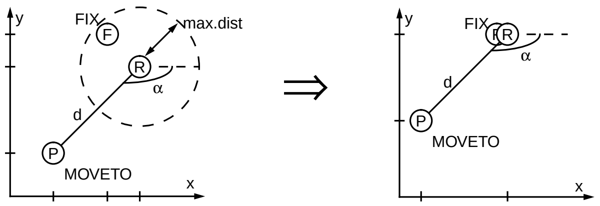


Fig. 34. Moveto marker updates its data based on the Fix marker when Fix is near enough. The ticks on the coordinate axis illustrate the coordinates of the marker points.

object point. The marker uses the same updating method as the Moveto marker. When the robot executes a transfer task, it turns towards the object at the approach point and pushes the object to the goal point. The Trajectory marker stores trajectories of moving objects, one at a time. A trajectory is a line fitted to a set of fixation points. The Sarea marker specifies a trajectory for the fixation point of the cameras.

The Zdfcent marker stores the center of the zero disparity region (the result of the ZDF). The rest of the markers store optical flow patterns. The patterns are labeled according to their role in the task by Home, Obstacle, and Robot markers.

The behaviors are presented in Table 4. The Lookfwd and Turn behaviors ensure that the robot looks around when there are no other active behaviors. The Back behavior drives the robot backwards for a while when the robot touches an object and the Transfer marker is not active. The Wait, Avoid, and Saccade behaviors operate based on image space markers. The Wait behavior stops the robot when an obstacle comes too near. The Avoid behavior avoids all obstacles. The Saccade behavior controls the cameras towards other robots.

Table 4. Pure behaviors.

Behavior	Type
Lookfwd, Turn, Back, Wait, Avoid, Saccade, Moveto, VisServo, Push, Pursuit, Search	primitive
Pose, Transfer, Receive, Unblock	task

The Moveto behavior moves the robot to the pose point of the Moveto marker. The VisServo behavior moves the robot towards the same point, but based on direct sensor data (that is, based on the Home marker). The Push behavior keeps the pushed object in front of the robot by controlling robot heading. The Pursuit behavior turns the cameras towards the Zdfcent marker. The Search behavior controls the fixation point of the cameras along the segment of the line specified in the Sarea marker.

The priorities of the behaviors controlling the cameras are defined according to their effect on the task execution. Among the basic behaviors controlling the robot, the behaviors preventing collisions have higher priorities than the ones executing tasks. Of two task executing behaviors, the one using direct sensor data has higher priority.

The Pose, Transfer, Receive, and Unblock behaviors and the Transfer marker form the task layer producing Moveto tasks for the reactive system. These tasks are communicated for the reactive system through the Moveto marker. The decomposition of the Pure system into the task and reactive layers is shown in Fig. 35.

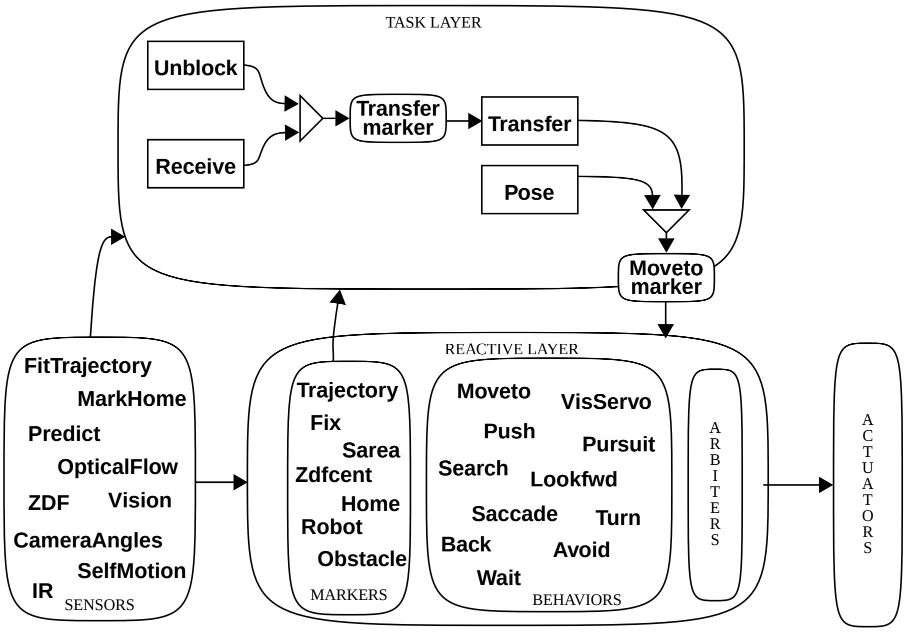


Fig. 35. The decomposition of Pure into the task and reactive layers. The triangles denote task arbiters.

The Pose behavior executes a posing task by initializing the Moveto marker. The Pose behavior is activated when the cameras are fixated on an object. The Transfer behavior executes a transfer task by controlling the Moveto marker according to the Transfer marker (see Fig. 36). The task arbiter controlling the Moveto marker prefers the Transfer behavior over the Pose behavior because a transfer task is more important than a posing task.

The Receive behavior executes a receiving task by controlling the Transfer marker. The Receive behavior is activated when the robot detects that another robot has released its cargo. The object point of the Transfer marker is associated with the object, the approach point is a predefined distance in front of the object and the goal point a predefined distance behind the object. The Unblock behavior executes an unblocking task by controlling the Transfer marker. The Unblock behavior is activated when an obstacle is found from the

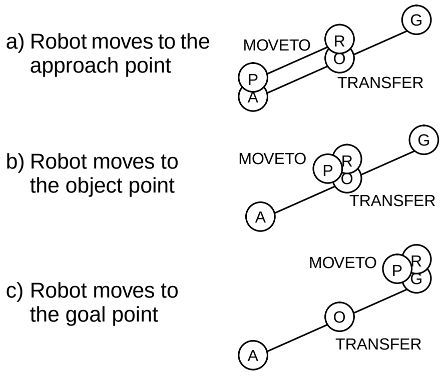

a) Robot moves to the approach point
MOVETO
TRANSFER
P
A
R
O
G
b) Robot moves to the object point
MOVETO
TRANSFER
P
R
O
A
G
c) Robot moves to the goal point
MOVETO
TRANSFER
P
R
G
O
A

Fig. 36. Moveto and Transfer markers during a transfer task.

trajectory of another robot. Obstacles are found by searching the trajectory of a posed robot periodically. A Transfer marker initialized by the Unblock behavior is perpendicular to the other robot's trajectory, whereas the marker initialized by the Receive behavior is parallel with the trajectory. As a receiving task is more important than an unblocking task, the task arbiter controlling the Transfer marker prefers the Receive behavior over the Unblock behavior.

The architecture also contains virtual sensors processing data. The MarkHome sensor associates the Home marker (used by VisServo) with the Moveto marker. The FitTrajectory sensor fits a line into the collected positions of another robot and stores the result in the Trajectory marker. The Predict sensor uses the trajectory to calculate the area in which an obstacle blocks the other robot. It places the result in the Sarea marker. These three modules are not behaviors, as they do not produce commands for the actuators.

When the experiments were performed, the system was capable of executing posing and unblocking tasks. The behaviors relying on optical flow calculation were separate control systems and not integrated in the Pure system. The image space buffer contained only a marker describing the region of zero disparity. All markers in the mobile space buffer were implemented. All behaviors connected to the mobile space buffer were implemented, except Receive and MarkHome, which use optical flow data.

4.2 Experiments

In the first experiment we tested the posing task. The robot kept a predefined distance to a moving target successfully for several minutes. Fig. 37 shows the robot in front of the target. The status of the mobile space buffer during posing is shown in Fig. 38. The Moveto marker is active. The reference point is associated with the target. In the figure, the robot is at the pose point.

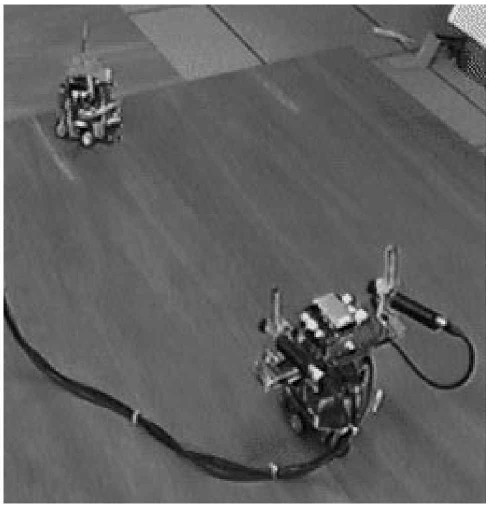
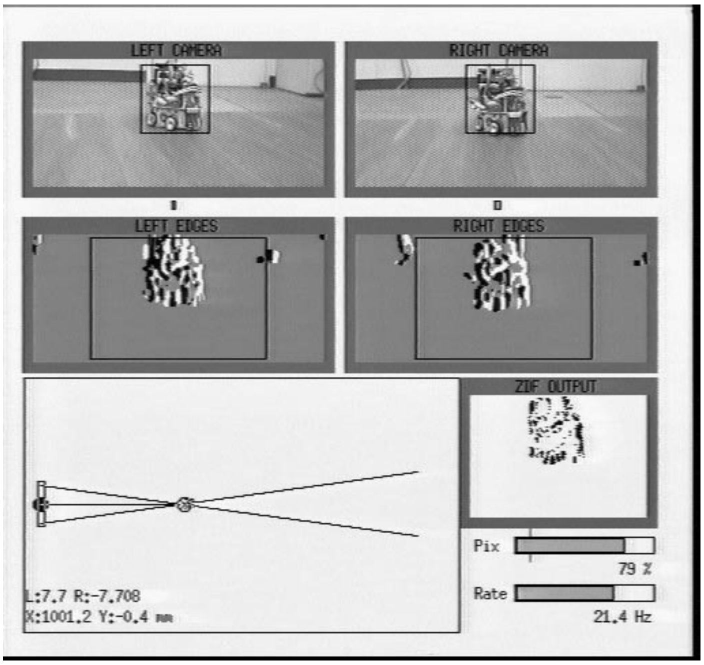


Fig. 37. The posing experiment.



L:7.7 R:-7.708
X:1001.2 Y:-0.4 mm

Fig. 38. Mobile space buffer during the posing experiment. The buffer is the area at the bottom left. The black circle is the pose point of the Moveto marker and the white circle the reference point. The images of the cameras are shown at the top. Below them are edge images. The output of the ZDF is shown below the right edge image.

Fig. 39 shows the system operation during the posing task. Several control loops are active in parallel during task execution. In the planning loop, the Pose behavior initializes the Moveto marker, which contains a point bound to an object and a pose point for the robot. The Moveto marker sends data to the Pose behavior. Generally, such a mechanism can be used to monitor the progress of a task. However, as the Pose behavior is a simple one, it only uses the data sent by the Moveto marker to monitor whether it has the control of the Moveto marker; whether it won the arbitration of the Moveto task.

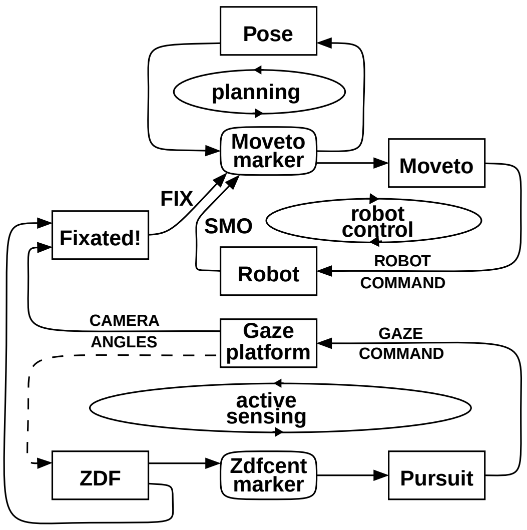


Fig. 39. System operation during a posing task. Planning, robot control, and active sensing loops are active in parallel. SMO means self motion.

The Moveto behavior controls the robot to the pose point and at the pose point turns the robot towards the reference point. The Moveto marker updates the pose and reference points continuously based on robot self motion. This is the robot control loop. Also the Pursuit behavior is active. It receives from the ZDF module the location of the posed object and controls the cameras towards it. The camera movements change the output of the ZDF module. This is the active sensing loop. Active sensing affects also the Moveto marker, through the Fix marker. All these loops are executed continuously at a frequency over 20 Hz

In the second experiment the robot helped another robot by pushing away an obstacle blocking the other robot. This is an example of multi-agent cooperation. The mobile space markers during an unblock task are presented in Fig. 40. Fig. 41 shows the mobile space buffer at the different stages of the unblock task. First the robot tracked another robot and calculated its trajectory. In the top figure, the robot is searching for an obstacle from the trajectory of the other robot. This situation is shown in Fig. 42.

After the obstacle was found, the Unblock behavior initialized an unblock task (Transfer marker), the Transfer behavior initialized the Moveto marker, and the Moveto

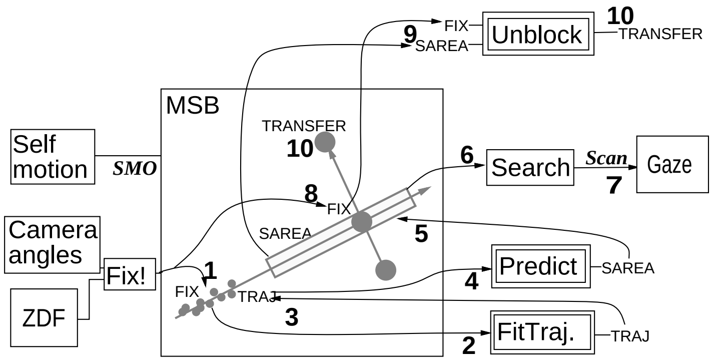


Fig. 40. Markers during an unblock task. The numbers mark the order of the events: (1) A set of fixation points is collected. (2-3) FitTrajectory calculates a trajectory. (4-5) Predict calculates a search area. (6-7) Search controls the fixation point along the search area. (8) An object is found. (9-10) The Unblock behavior initializes a transfer task.

behavior started to control the robot towards the approach point. The middle figure of Fig. 41 shows the mobile space buffer when the robot has just reached the approach point of the Transfer marker. After this, Transfer initialized two new moveto tasks, one to the object point and one to the goal of the transfer. Finally, the bottom figure of Fig. 41 shows the mobile space buffer when the robot is reaching the goal.

4.3 Discussion

The experiments on the Pure architecture taught us the importance of grounding everything physically. Updating even the simple representations, mobile space markers, turned out to be difficult. This is because the sensor data is uncertain. The mobile space markers are updated based on the fixation point, which is calculated from the camera angles. Because of short camera baseline and unreliable angle measurements, the calculated distance from the robot contains a considerable error, specifically at the sides of the field of view where the camera directions are nearly parallel. The mobile space markers are updated also by robot motion. Unreliable motion measurements and the wires connecting the robot to the host computer cause considerable error in calculated motions. As a result, it was difficult to represent the task related environment features with the accuracy required by a successful task execution.

Although accuracy could doubtlessly be improved by describing uncertainties explicitly and filtering, we chose to first relax the requirements of accuracy. Accurate data is needed because the robot must perform accurate operations. If the sensors can be directed towards the task related environment features during the task execution, the

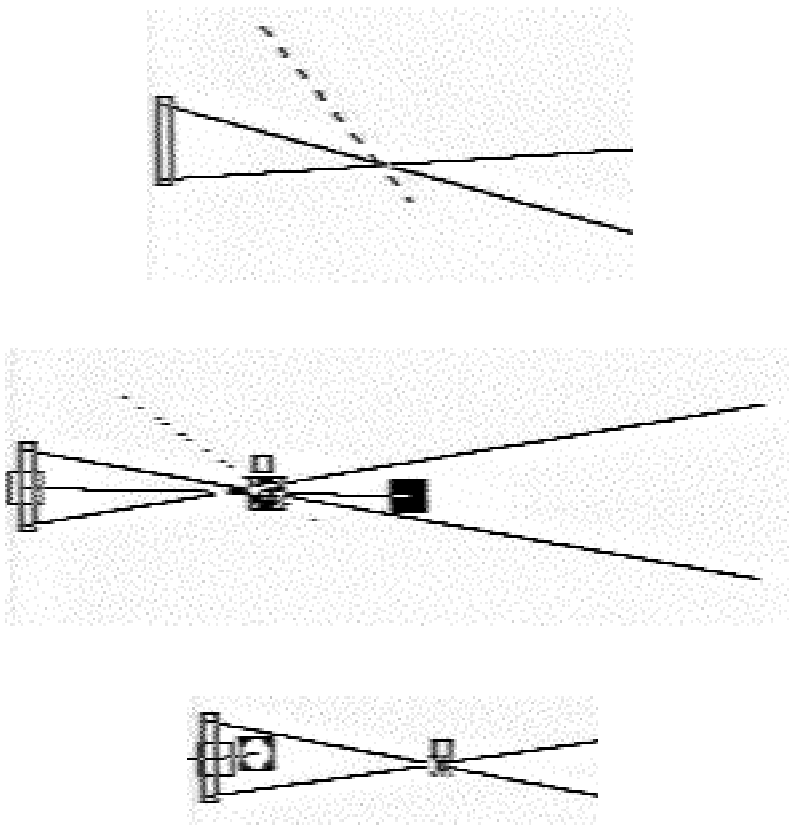


Fig. 41. Mobile space buffer at the different stages of the unblocking experiment. Top: the robot is searching an obstacle around the trajectory of the other robot (the dotted line). Middle: the robot has just reached the approach point of the Transfer marker. The black box at the right is the goal point, the box in the middle is the object point, and the box aligned with the robot is the approach point of the Transfer marker. Bottom: the robot is pushing the object towards the goal. The box aligned with the robot is the object point and the box to its right is the goal point of the Transfer marker.

accurate values can be perceived and the requirements for the accuracy of the representation can be relaxed.

How, then, can the sensors be directed towards the correct features? The Pure architecture already contains a mechanism for focusing attention, namely the mobile space markers. When a task is to be executed, the attention of the robot can be focused on the right environment feature based on the corresponding mobile space marker. The marker does not have to represent the properties of the feature accurately at every moment. Instead, it suffices that the feature can be found from the environment based on the data stored in the marker. When attention is focused on the feature in question, accurate values are perceived and hence accurate operations can be performed.

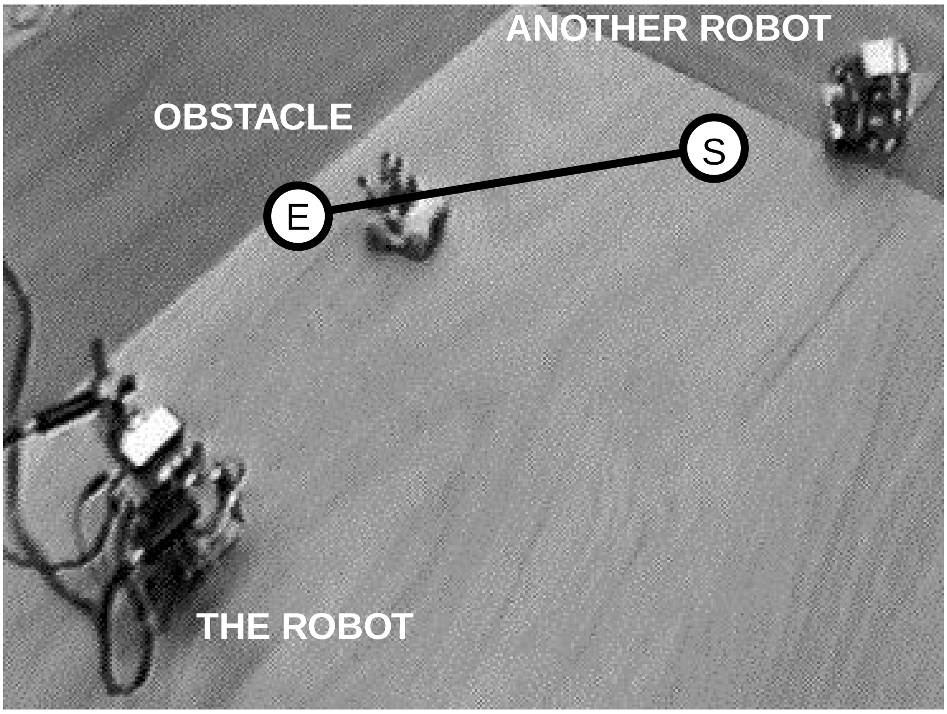


Fig. 42. The controlled robot executing an Unblocking task. The Search behavior is controlling the fixation point of the cameras along the Search marker. The start point of the marker is marked by an S and the end point by an E.

From this discussion, we suggest that markers need not describe the exact values of the task related environment features, but directions on how to perceive these features. Hence a marker can be defined as an index to a feature. This definition leads to the following constraint: a marker is a processing element associating a *perceivable* environment feature with task related data.

This constraint solves the problem of unreliable fixation point calculation. As the erroneous locations are used only to direct the cameras towards the right feature, larger errors are tolerable. Also the errors in robot motion can be corrected by perceiving the feature indexed by the marker, as all markers are perceivable.

The discussion above applies only to the mobile space markers, as all image space markers are always related to direct sensor data. Mobile space markers focus attention towards the task related features. The image space markers label these features. The behaviors control the robot based on the image space markers that have the accuracy required for accurate operations.

In this framework all features associated with mobile space markers must be perceivable. This is a strong constraint. Past or future locations of objects can be stored in the mobile space buffer only as fields of an object marker. This is the case with the

Trajectory marker. Fitted trajectories can be stored in markers associated with moving objects. The Search area can still be stored in the mobile space buffer, but it has to be connected to a marker associated with a perceivable feature. Thus, knowledge about the focus of attention can be shared among behaviors.

In other words, our experience leads us to suggest that a stored representation of the environment should be interpreted as directions for perceiving the corresponding values directly from the environment. That is, the stored representation indexes an object in the environment. Markers can be used as such a representation.

Another problem related to uncertain and varying sensor readings is that of activating behaviors. As many behaviors are activated on the basis of thresholds defined in terms of sensor readings, the robot easily oscillates between behaviors when the sensor readings vary around thresholds. We experienced this problem when testing the Moveto behavior. The Moveto behavior oscillated between driving towards the goal and staying at the goal as the distance to the goal fluctuated. We solved this problem by hysteresis. Two features become equal when the distance between them decreases below a minimum threshold. The equality is remembered. The features stay equal as long as the distance does not exceed a maximum threshold (see Fig. 43).

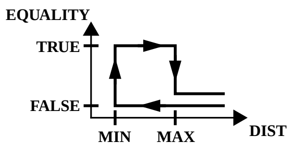


Fig. 43. The equality of features has different thresholds depending on the direction of change.

The improved Moveto behavior uses minimum and maximum thresholds to decide when the robot is at the goal. When the distance decreases below the minimum threshold, it stops the robot. If the distance exceeds the maximum threshold, it starts to control the robot towards the goal again.

The experiments showed that markers improve a behavior-based system. Markers simplify behaviors, as behaviors need not update task related data themselves. Markers also focus the attention of the behaviors and thus constrain the situations in which the behaviors operate. Further, markers decrease the number of connections between the control system modules. A behavior does not need to be connected to all modules updating the data it uses. Instead, it suffices to connect the behavior to the corresponding marker. Markers also increase the modularity of the control system. The behaviors are grouped into sets performing tasks. A further advantage is that conditions for tasks are easy to express using markers and their spatial relations. Finally, all these improvements were achieved without sacrificing the real time performance of a behavior-based system.

5. Samba for Soccer: experiments on action maps

In this chapter, we describe the second set of experiments. In these experiments, we applied the Samba architecture to playing soccer in a simulated environment. We tested the complete Samba architecture containing both markers and action maps. As was the case with the first set of experiments, the primary goal was to show how markers and action maps facilitate the fulfillment of the requirements for reactive task execution. Furthermore, the aim was to prove that these extensions do not destroy the reactiveness of a behavior-based system. A secondary goal was to illustrate the properties of the control architecture.

The Samba for Soccer architecture controls a soccer player. It controls a player to gain possession of the ball, to move it by dribbling and passing nearer the opponents' goal and to score goals. All these actions are produced by combining primitive action maps. The first set of experiments was run in a simple simulator containing the player, the ball, some team-mates, some opponents, and one goal. The second set of experiments was run in the Soccer Server, a soccer simulator offered by the RoboCup Committee (Kitano *et al.* 1997). In these experiments, 11 copies of a Samba for Soccer control system were connected to the Soccer Server. This team, CAT Finland, played against the teams developed by other robotic research groups. These experiments were performed both at the University of Oulu and at the Robot Soccer World Championships, in 1997 at Nagoya, Japan (RoboCup'97) and in 1998 at Paris, France (RoboCup'98).

The Soccer Server simulates a soccer field, a referee, a ball and two teams, each consisting of 11 players. A player in the Soccer Server observes the locations and velocities of the other objects in front of it and executes turn, dash, kick, and shout actions. The observed objects are the ball, the other players, the goals, the borders, and the flags. All information is relative to the player. A player also hears the messages shouted by the other players and the referee. The energy resources of a player are limited. The stamina of a player decreases when the player runs. The rate of decreasing the stamina depends on the running speed. The stamina increases according to the recovery rate when the player rests. If the stamina is diminished low enough, the recovery rate of the player decreases permanently.

We decided to use soccer as an application for several reasons. First, we are developing a control architecture for mobile robots that operate in a dynamic environment and cooperate with other agents (robots and humans). Soccer is an excellent testbed for such robots. Second, the Soccer Server simulator offered by the RoboCup Committee enabled us to test the architecture right away instead of building one more simulator first. Finally, in the RoboCup tournaments we can compare our control approach with the approaches of the other research groups.

5.1 Architecture

The Samba for Soccer architecture together with the task layer is shown in Fig. 44. The Vision sensor produces the locations and velocities of all the objects on the player's visual field of view: the ball, the team-mates, the opponents, the opponents' goal, the home team's goal, the borders, and the flags. In addition to the flags seen in the real soccer field, there are extra flags around the field serving as landmarks. All the information is relative to the player. The Hearing sensor listens to the messages shouted by the other players and the referee. The Body sensor reports the stamina and the speed of the player. It also reports the number of the executed commands which can be used in approximating which of the sent commands have been executed.

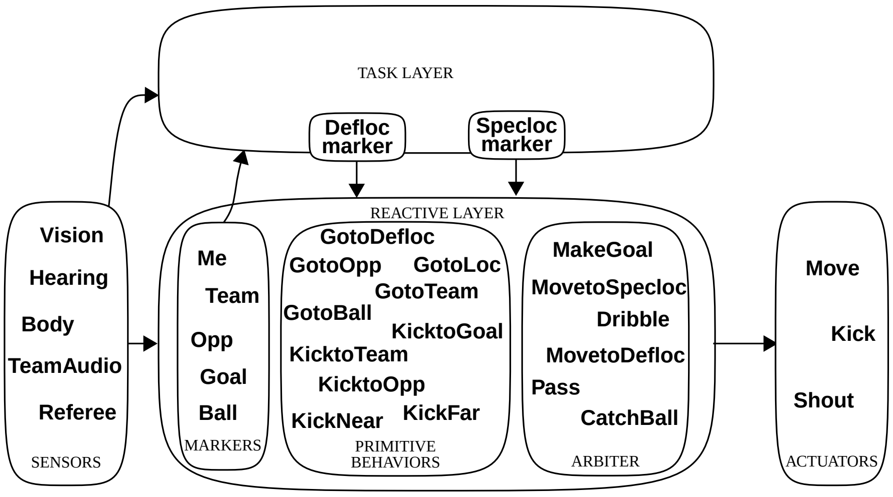


Fig. 44. Samba for Soccer. A primitive behavior produces a map for both reaching and avoiding an object (or a location). The action maps compiled by the arbiter are listed inside the arbiter module. Due to the large number of connections between the modules, the connections are not drawn in the figure. The modules are connected to each other as in Fig. 11.

The Referee virtual sensor reports the messages shouted by the referee. The referee informs the players about the state of the game and gives kick ins (they replace throw ins), free kicks, corner kicks, and goal kicks. The TeamAudio virtual sensor reports the messages shouted by the team-mates. The Move actuator executes the turn and dash commands, the Kick actuator the kick commands, and the Shout actuator shouts messages to other players.

The markers are shown in Table 5. The system contains markers for the opponents, the ball, the team-mates, the opponents' goal, the home team's goal, and the player itself. The team-mate (Team) and opponent (Opp) markers represent the locations and velocities of all team-mates and opponents whose locations are known. The Goal marker describes the locations of both of the goals relative to the player.

Table 5. The Markers.

Marker	Type
Me, Ball, Team, Opp, Goal	object
Defloc, Specloc	task

The Me marker describes the location and velocity of the player itself. It is used when taking the communication and processing delays into account. The marker describes the state of the player at the moment of perceiving the latest sensor data. In addition, it describes the motion the player will perform between this moment and the moment the execution of the command which is being calculated starts. The location of the player on the field is calculated from the observed flags. When updating the markers with the latest sensor data, the locations of the objects are projected using the Me marker to the moment the execution of the command which is being calculated starts. Hence, no other modules need to reason about the delays.

The task markers form an interface between the Samba system (i.e., the reactive layer) and the task layer. As this system plays reactive soccer, there are only two task markers, the Specloc marker and the Defloc marker. Both markers are updated by the task layer. When compared to updating by object markers this updating method results (in this case) in a simpler system without introducing extra delays.

When a player is to take a free kick, a goal kick, a corner kick, or a kick in, the task layer initializes a task to move to a good location to perform the kick. This location is specified by the Specloc marker. The task layer controls the location of the player through the Defloc marker. The player moves to the default location specified by this marker when it does not have more important tasks. The task layer computes the default location based on the role of the player and the location of the ball. For example, the default location of an attacker might move from the halfway line towards the opponents' goal when the ball moves from the home team's goal towards the opponents' goal. The location of the player could be controlled also according to the opposing team. An example of the default locations of the players are shown in Fig. 45.

As the control system controls the movements of both the player and the ball, the system produces two types of action maps. A velocity map specifies preferences for the different actions for reaching an object. An impulse map presents weights for all the

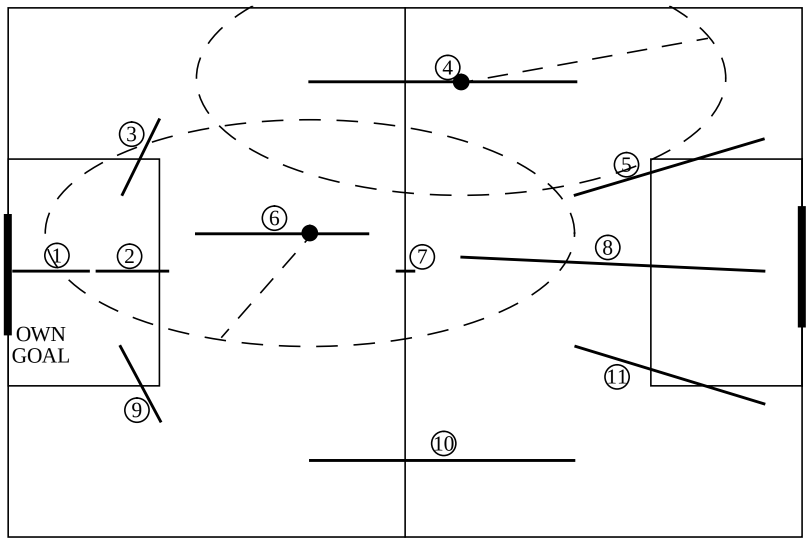


Fig. 45. Default locations of the players. The default location of a player moves along a line based on the location of the ball. The ellipses define for players 4 and 6 the allowed operation areas when the default locations are at the locations marked by the black circles.

possible different impulses (direction, magnitude) the player can give to the ball. It specifies the preferences for the different actions for kicking the ball to reach an object. Table 6 presents the primitive maps. Most of the Avoid and DontKickto maps are calculated by negating the corresponding Goto and Kickto maps.

Table 6. The primitive maps.

Map type	Primitive velocity maps	Primitive impulse maps
Goto	GotoBall, GotoTeam, GotoOpp, GotoLoc, GotoDefloc	KicktoGoal, KicktoTeam, KicktoOpp, KickNear, KickFar
Avoid	AvoidBall, AvoidTeam, AvoidOpp, AvoidLoc, AvoidDefloc	DontKicktoGoal, DontKicktoTeam, DontKicktoOpp, DontKickNear, DontKickFar

The GotoLoc and GotoDefloc maps are calculated by the projecting method for a point. The other primitive Goto maps, GotoBall, GotoTeam, and GotoOpp are calculated by the projecting method for a circular object. The weight curve utilized in calculating velocity maps is shown in Fig. 46. The actions requiring a long time are given small weights to increase the number of progressing actions.

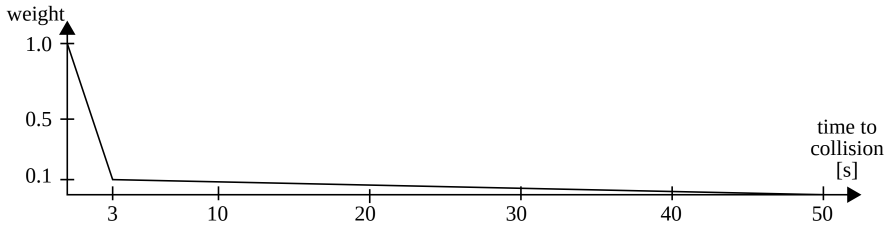


Fig. 46. The weight curve utilized in calculating velocity maps. All actions lasting 0.1 seconds or less get the full weight of 1.0.

The KicktoGoal map contains heavy weights for the maximum impulses that direct the ball towards the opponents' goal. The KicktoTeam map contains heavy weights for all impulses that result in a collision between the ball and the team-mate in question. The team-mate is supposed to keep the observed velocity until it reaches the ball. The KicktoOpp map contains heavy weights for all impulses that cause the opponent to reach the ball. The opponent is supposed to try to gain possession of the ball as fast as possible. The weights for the impulses directing the ball towards the opponent are calculated according to the distance to the opponent, the predefined reaction time, and the predefined speed of the opponent. The rest of the weights are approximated by propagating. The KickNear map contains heavy weights for those impulses causing the ball to reach a predefined distance towards the opponents' goal in a predefined time. The map is propagated to produce positive weights for a wide sector. The KickFar map contains heavy weights for the impulses directing the ball towards the opponents goal at a high speed. The map is propagated to produce positive weights for a wide sector. The weight curve utilized in calculating impulse maps is otherwise similar to the one used for velocity maps, but the weight decreases from the value of 1.0 slightly faster when the collision time increases.

The primitive behaviors calculate separate action maps for all opponents and team-mates represented by the markers. The AvoidOpp and AvoidTeam maps contain heavy weights only for the actions causing a collision, because in soccer it is important not to avoid other players too much. This is accomplished by calculating weights only for the actions in the object sector and by not projecting the actions in the projecting sector (see Fig. 25).

The GotoLoc map is utilized when a player is to take a free kick, a goal kick, a corner kick, or a kick in. It controls the player towards the Specloc marker. The GotoDefloc map controls a player to its default location specified by the Defloc marker. A filter mask is used to remove from the impulse maps the impulses that would cause the ball to collide with the player kicking the ball. This ForbiddenSector mask contains a zero for each action to be removed and one for the rest of the actions. Fig. 47 shows an example of a ForbiddenSector mask.

The composite maps compiled by the arbiter are listed in Table 7. Only one team-mate and opponent map of a type is listed for each composite map, although all team-mate and opponent maps are utilized. The MovetoSpecloc map controls the player towards a good location (special location) when a goal kick, a free kick, a kick in, or a corner kick is to be

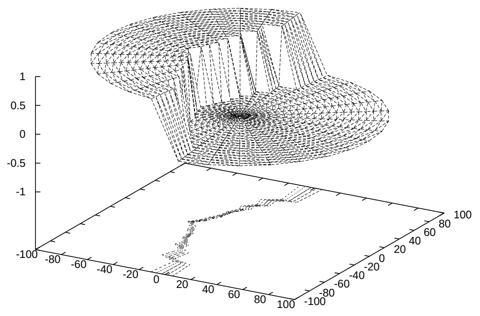


Fig. 47. A forbidden sector mask for a situation in which the ball is behind the robot and moving towards the robot. To prevent a collision, the robot should kick the ball backwards hard enough.

taken. In this map, the AvoidBall map prevents only actions that would result in a collision with the ball in less than 1.5 seconds. The MakeGoal map contains heavy weights for impulses directing the ball towards the goal but avoiding the team-mates and the opponents. The task of scoring a goal is executed by modifying and combining the primitive reactions to team-mates, opponents, and the opponents' goal.

Table 7. The composite maps.

MovetoSpecloc = MAV(GotoLoc, AvoidOpp, AvoidTeam, filter(AvoidBall, 1.5))
MakeGoal = filter(MAV(KicktoGoal, DontKicktoOpp, DontKicktoTeam)), ForbiddenSector)
Pass = filter(MAV(KicktoTeam, DontKicktoOpp), ForbiddenSector)
Dribble = filter(MAV(KicktoNear, DontKicktoTeam, DontKicktoOpp), ForbiddenSector)
CatchBall = MAV(GotoBall, AvoidOpp, AvoidTeam)
MovetoDefloc = MAV(GotoDefloc, AvoidOpp, AvoidTeam)

The Pass map contains heavy weights for impulses causing the ball to collide with a team-mate while avoiding the opponents. The Dribble map favors impulses directing the ball towards the opponents' goal at a low speed, avoiding team-mates and opponents.The CatchBall and MovetoDefloc maps have heavy weights for velocities that direct the player towards the ball and the default location, respectively. Both maps have low weights for velocities that cause collisions with team-mates and opponents.

Each of the composite impulse maps is filtered with the ForbiddenSector mask. Another possibility would be to consider the forbidden sector when creating the primitive maps.

Filtering was chosen because the player might need to find the best action, regardless of whether it results in a collision or not. For example, if the action map reveals that the player has a very good chance of scoring a goal, it might decide to try even when the ball is behind; it might first move the ball in front and then try to score a goal. Furthermore, the selected approach results in simpler algorithms, as the forbidden sector does not need to be considered when creating the primitive maps.

A task is performed by sending the action with the heaviest weight in the appropriate composite map to the actuators. The task to be executed at each moment is selected by the arbiter, which goes through the tasks in the following order: SpecialSituation, LookAround, MakeGoal, Pass, Dribble, CatchBall, MovetoDefloc, and TurntoBall. The arbiter selects the first task whose valid conditions are fulfilled. Then the arbiter compiles the composite map corresponding to the task, finds the best action in the map, and sends it to an actuator. When a task has been selected, it remains active as long as its valid conditions are fulfilled. Among the tasks listed above, the LookAround and TurntoBall are executed without utilizing action maps. The SpecialSituation task expands into a prioritized task set similar to the list above. This task set contains the MovetoSpecloc task listed in Table 7.

These task sets and the valid conditions of the tasks produce the following behavior: as long as a special situation (a free kick, goal kick, etc.) is observed, the player performs the SpecialSituation task. Otherwise, if the location of the ball or the location of the player is not certain the player looks around until the certainty increases to an acceptable level. Otherwise, if the ball is near enough, the player kicks it. It favors scoring a goal. If the weight of the best action to score a goal is too low, it considers passing the ball to a team-mate. If this cannot be done either, the player dribbles the ball. If the ball is not near enough to be kicked, but no team-mate is nearer the ball, the player tries to gain possession of the ball. If a team-mate is nearer the ball and the player is too far from its default location, it runs towards it. Otherwise, the player turns towards the ball.

To prevent the performance of the player degrading, the maximum speed of the player is controlled according to the stamina. As the recovery rate has a significant effect on the behavior of the player, the most important energy saving rule is to never let the recovery rate decrease. This is accomplished by decreasing the maximum speed when the stamina approaches the threshold for decreasing the recovery rate. The maximum speed is calculated as a function of stamina, as illustrated in Fig. 48. The function has hysteresis: the value the maximum is set to depends on the direction of change of the stamina. As a result, the player rests for a while after becoming exhausted. The function for controlling the maximum speed is stepwise, as changing the maximum speed frequently would prevent predicting the results of actions. For example, reaching a moving object successfully requires that the maximum speed does not change while the player runs towards the predicted collision point. Another method of saving energy would be to prioritize the tasks and to relate the maximum speed of the player to the priority of the task under execution. However, such more complex methods were not necessary, as the control function discussed above keeps the stamina and thus the maximum speed near the maximum values.

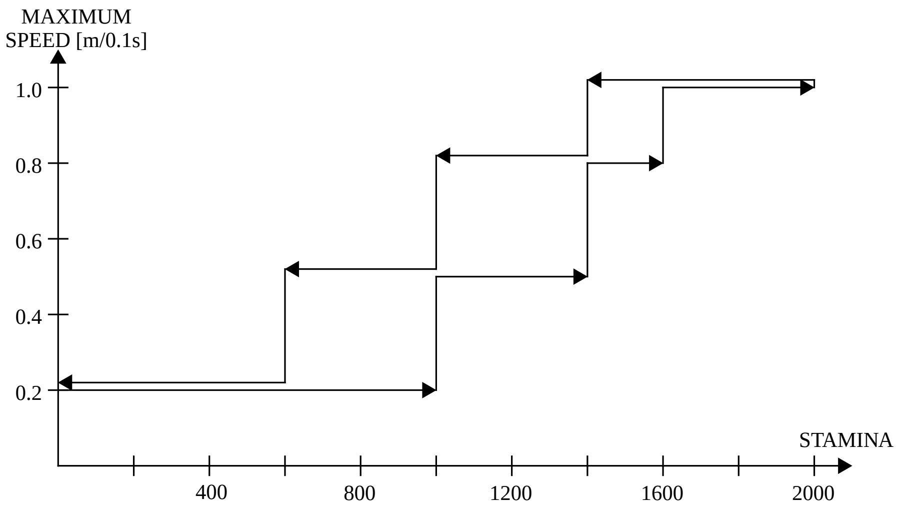


Fig. 48. The maximum speed of the player as a function of the stamina. The value the maximum is set to depends on the direction of change of the stamina. The maximum value of the stamina is 2000 units and the recovery rate starts to decrease when the stamina decreases below 600 units.

5.2 Experiments

In the first set of experiments, we ran the Samba for Soccer control system in a simple simulated environment. The player controlled by the Samba system, the team-mates, and the opponents were circular. At the beginning of each experiment, the player was heading to the right. The control system and the simulator were executed in turns. First, the control system calculated an action based on the sensor data. The sensor data described the positions and velocities of all objects in the environment. After that, the simulator moved the player according to the action and the objects based on their predetermined trajectories. The system was executed at a frequency of 10 Hz, so the simulator moved at each turn the player and other objects by the amount they move in 0.1 seconds.

We ran the following experiments. First, we tested reaching stationary and moving team-mates when there were no opponents to be avoided. We tested both reaching a team-mate moving at a constant velocity and reaching a team-mate that changed its velocity during the experiment. Second, we added both stationary and moving opponents, repeating the reaching experiments. This time the task was to reach the ball. The OSA method was utilized in calculating the velocity maps. Third, we tested the impulse maps by passing the ball to both stationary and moving team-mates and by scoring a goal.

In the first experiment, the player was given a task to reach a stationary team-mate. The trace of the experiment and the first two GotoTeam action maps are shown in Fig. 49. The GotoTeam action map on the left causes the player to turn towards the team-mate. After the player had turned, the control system produced the GotoTeam map shown on the right. In this map, the action of driving forward with the maximum speed has the heaviest weight. The control system produced similar maps until the player reached the team-mate. The maps shown in Fig. 49 reveal the form of the weight curve: weights decrease rapidly when the time required to reach the team-mate exceeds 0.1 seconds and actions requiring more than 3.0 seconds have very low weights.

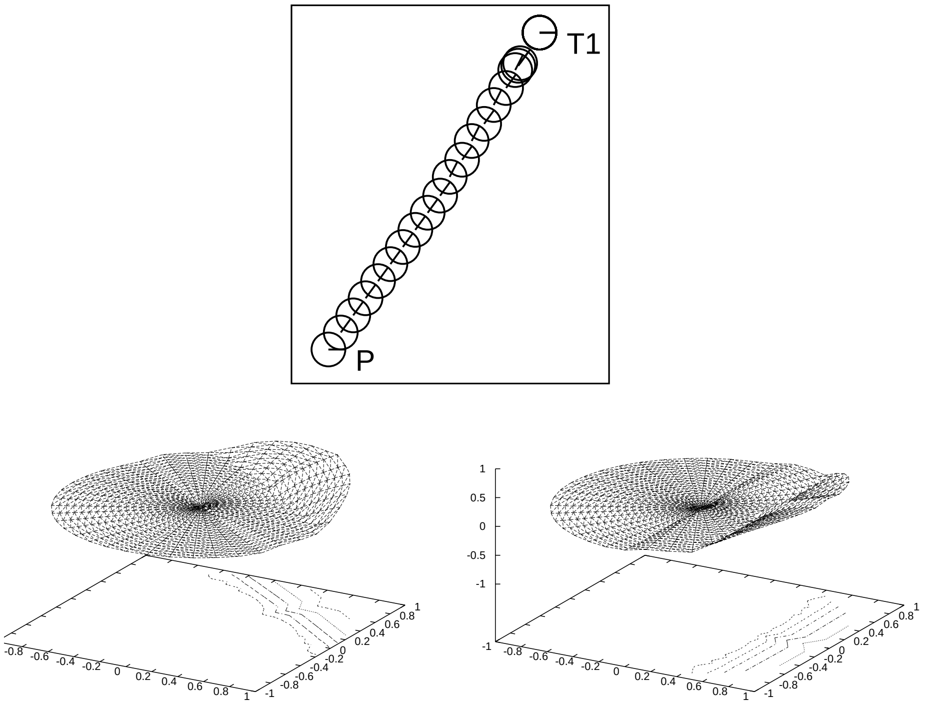


Fig. 49. The player (P) reaches a static team-mate (T1). Top: the trace. The initial distance to the team-mate is 18 meters. The segments of line show the forward directions of the players. Bottom left: the GotoTeam action map calculated at the initial situation. Bottom right: the GotoTeam map after the player has turned to the optimal direction.

In the second experiment, the player reached a moving team-mate. The trace of the experiment and the first two GotoTeam action maps are shown in Fig. 50. As was the case with the stationary team-mate, the first map turns the player to an optimal direction and the following maps keep that direction until the team-mate is reached. As the velocity of the team-mate does not change during the experiment, the player does not have to turn again. It drives forward and keeps the proper speed for reaching the team-mate. In all the maps, the action with the heaviest weight has a speed that cause the player and the team-mate to reach the collision point simultaneously.

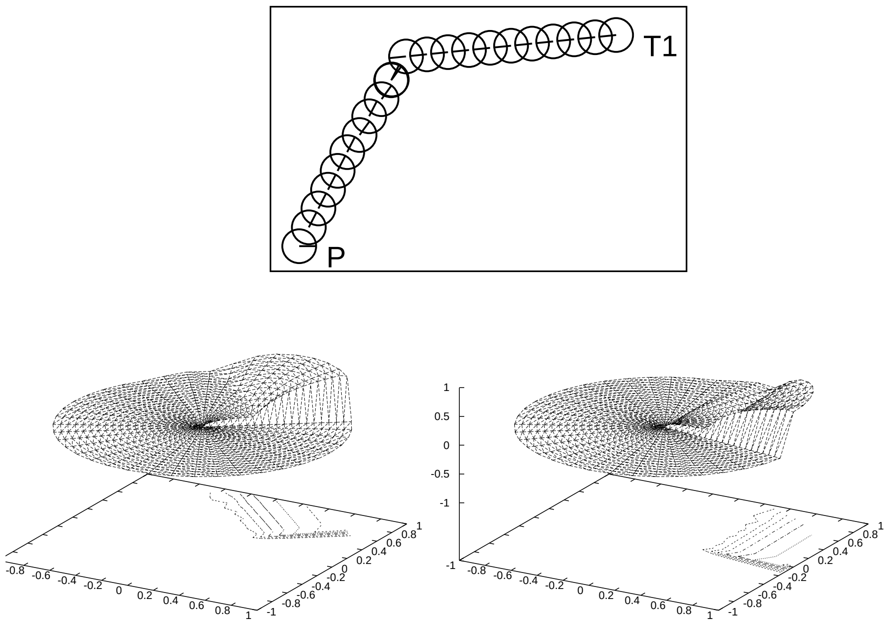


Fig. 50. Top: the trace of the experiment on the player (P) reaching a moving team-mate (T1). Bottom left: the GotoTeam action map calculated at the initial situation. Bottom right: the GotoTeam map after the player had turned to the optimal direction.

Fig. 51 shows traces of situations in which the team-mate changes its velocity after the player has turned towards the optimal direction to reach the team-mate. In the situation on the left in Fig. 51, the team-mate changes its direction, and in the situation on the right the team-mate changes its speed. As the player calculates action maps continuously, these changes do not cause any difficulties. As soon as the new velocities have been observed, the player turns to the updated optimal direction.

In the following experiment, the task was to reach a stationary ball located in front of a stationary opponent. Fig. 52 shows the situation and the AvoidOpp and CatchBall maps. The AvoidOpp map forbids all actions driving the player towards the opponent and hence towards the ball. But as the MAV method selects for each action a weight corresponding to the object that would be reached first, the AvoidOpp map has no effect. As a result, the player has no problems in reaching the ball.

Experiments on obstacle avoidance are shown in Fig. 29 and Fig. 30. In both experiments, a stationary obstacle is located between the robot and a stationary target. A Goto map is calculated for the target and an Avoid map for the obstacle. Fig. 29 shows the trace and the map compiled from the Goto and Avoid maps when the Avoid map is not filtered. All the actions in the direction of the target are prevented and hence the robot

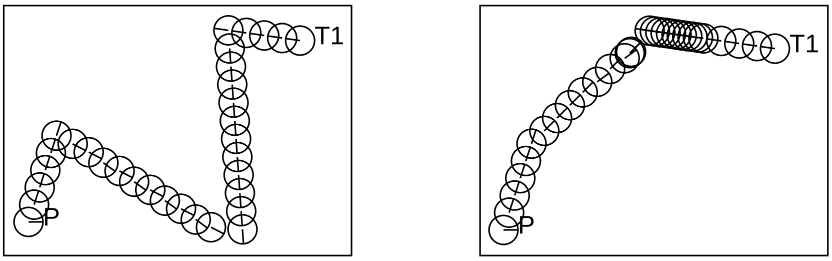


Fig. 51. Traces of experiments in which the team-mate (T1) changes its velocity after the player (P) has turned towards the optimal direction. Left: the team-mate changes its direction. Right: the team-mate changes its speed.

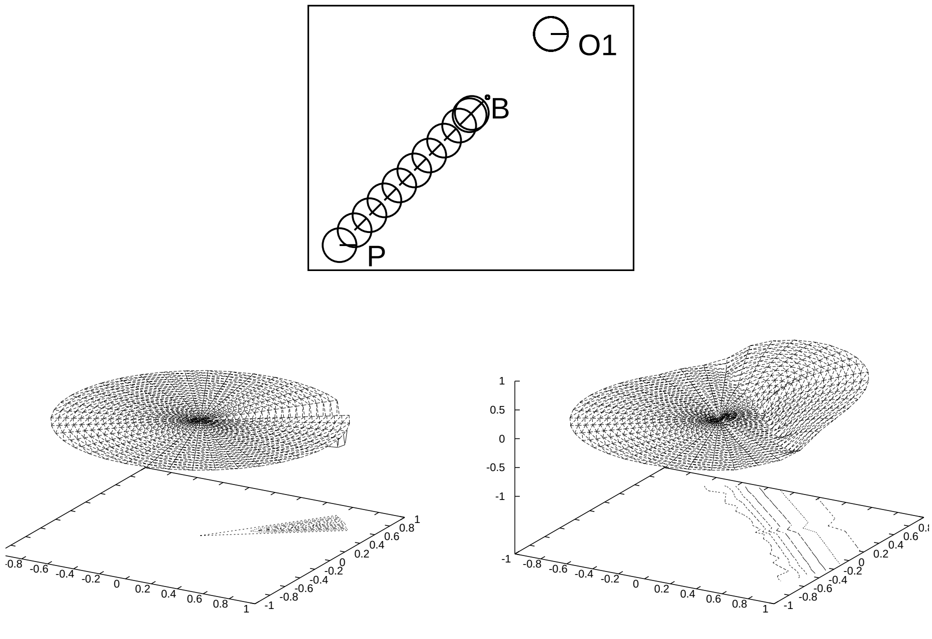


Fig. 52. The player (P) reaches a stationary ball (B) located in front of a stationary opponent (O1). Top: the trace. Bottom left: the AvoidOpp map. Bottom right: the CatchBall map reaching the ball and avoiding the opponent.

starts to circumvent the obstacle right away. Fig. 30 shows the same experiment with filtering. All actions requiring more than 0.5 seconds for a collision are filtered from the Avoid map. As long as the robot is far enough from the obstacle, all the actions in the

Avoid map are filtered and the robot moves straight towards the obstacle. When the time to reach the obstacle with an action goes below the filtering constraint, in this case below 0.5 seconds, the Avoid map prevents the action. As a result, the robot starts to circumvent the obstacle.

Fig. 53 shows comparison of the MSA and OSA methods in a situation in which a ball moves in the middle of stationary opponents. The maps in the middle row are calculated

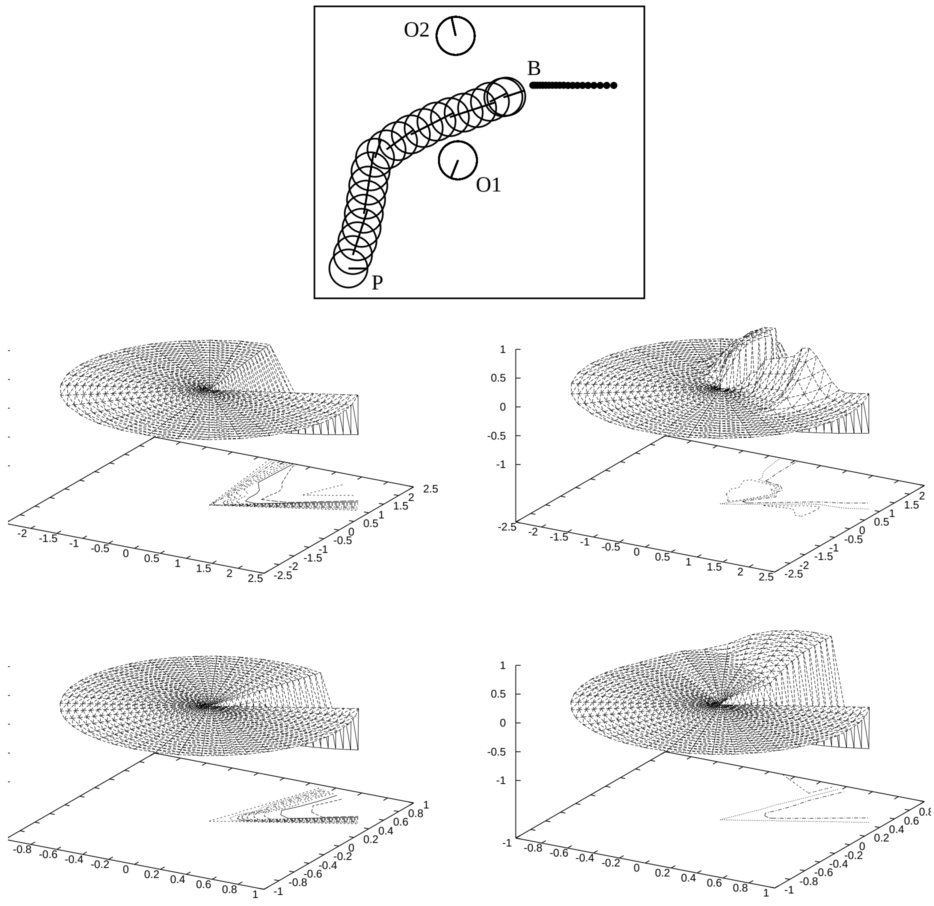


Fig. 53. Reaching a moving ball in the middle of stationary opponents. Top: the trace for the OSA method. Middle row: maps calculated by the MSA method, AvoidOpp on the left and CatchBall on the right. Bottom row: maps calculated by the OSA method, AvoidOpp on the left and CatchBall on the right.

utilizing the MSA method. The CatchBall map encodes the situation correctly: the opponent behind the trajectory of the ball does not prevent any actions in the GotoBall map, but the opponent in front of the trajectory does. This is achieved by calculating for each possible alternative action a weight based on the collision time for that action. However, the actions with high speeds are forbidden in the direction of the opponent behind the trajectory. This is because the GotoBall map does not contain heavy weights for these actions, as the player would reach the collision point too quickly and miss the ball were one of those actions executed.

The maps in the bottom row were calculated with the OSA method. The opponent behind the trajectory blocks the actions in that direction. This is because the weight is calculated according to the collision time only for the optimal action. For the rest of the actions, the weight is calculated by projecting. The opponents far enough do not block any actions. The maximum distance for an opponent blocking actions depends on the angular distance between the optimal direction and the opponent.

In the optimal direction, the combination produces similar results with the MSA method; the situation is encoded correctly in the resulting map. The OSA method does not produce as good a result in other directions as the MSA method, simply because it considers only the optimal direction. Furthermore, the player avoids the opponent behind the ball’s trajectory only at the beginning of the experiment. After the player has moved a small amount, the sectors covered by the opponents do not touch each other anymore and the player turns to a direction between the opponents.

In Fig. 54, the ball is moving behind a moving opponent. The CatchBall map in the initial situation is shown in the same figure. The map is compiled from a GotoBall and an AvoidOpp map. As these primitive maps encode the dynamics of the objects correctly, and the MAV method preserves the information in these maps, the combined map also encodes correctly the dynamics of the situation. As a result, the player drives straight towards the optimal collision point behind the opponent as the opponent will move away before the player reaches it.

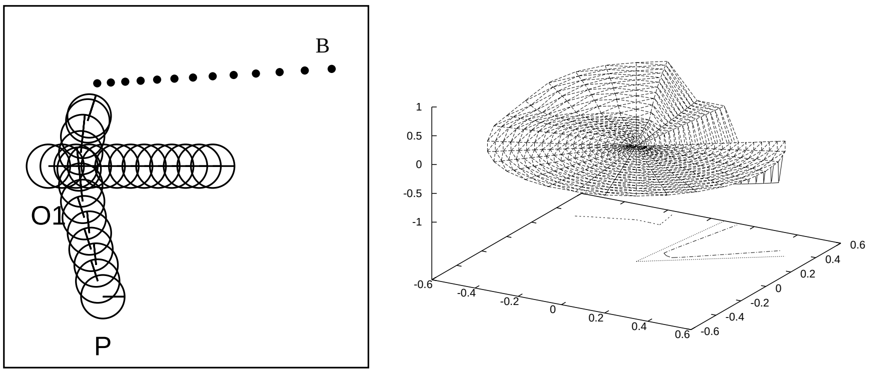


Fig. 54. The player (P) reaches a moving ball (B) in spite of a moving opponent (O1) blocking the optimal direction in the initial situation. Left: the trace. Right: the CatchBall map compiled from a GotoBall and an AvoidOpp map in the initial situation. The opponent does not inhibit actions in the GotoBall map.

We started the experiment on impulse maps by passing the ball to a stationary team-mate. Fig. 55 shows the situation and three KicktoTeam impulse maps calculated in the situation: one for a stationary ball, one for a ball moving to the right at the speed of one meter per second, and for a ball moving to the left at the same speed. This experiment illustrates the effect of ball momentum on the impulse maps. The speed of the ball at the

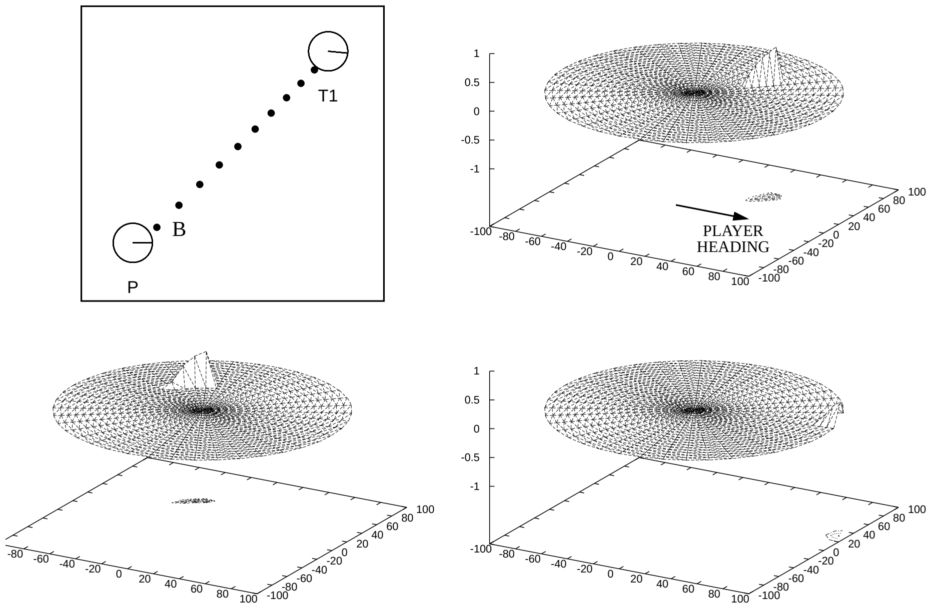


Fig. 55. The player (P) is to pass the ball (B) to a stationary team-mate (T1). Top left: the trace for the initially stationary ball. Top right: the KicktoTeam impulse map for a stationary ball. Bottom left: the KicktoTeam map for a ball moving at the speed of one meter per second to the right. Bottom right: the KicktoTeam map for a ball moving at the speed of one meter per second to the left.

moment of reaching the team-mate is constrained to be at most five meters per second. This constraint, the maximum speed on reaching its destination, causes the strong impulses to be rejected in the first two cases. In the third case, the map contains only strong impulses, as the difference between the ball’s momentum and the direction of the team-mate is considerable.

Fig. 56 illustrates passing a ball to a moving team-mate. The actions with a positive weight form a narrow ridge on the map, because the conditions for a successful task are strict: the ball is to be at the right location at the right time. This experiment illustrates also the effect of the maximum speed on reaching its destination. The maps calculated for maximum speeds of three, five, and eight meters per second are quite different. Particularly, it can be noted that setting the maximum speed to a small value limits considerably the number of alternative actions. This is because the impulse given to the ball cannot be calculated based on the speed allowed on reaching its destination. Instead,

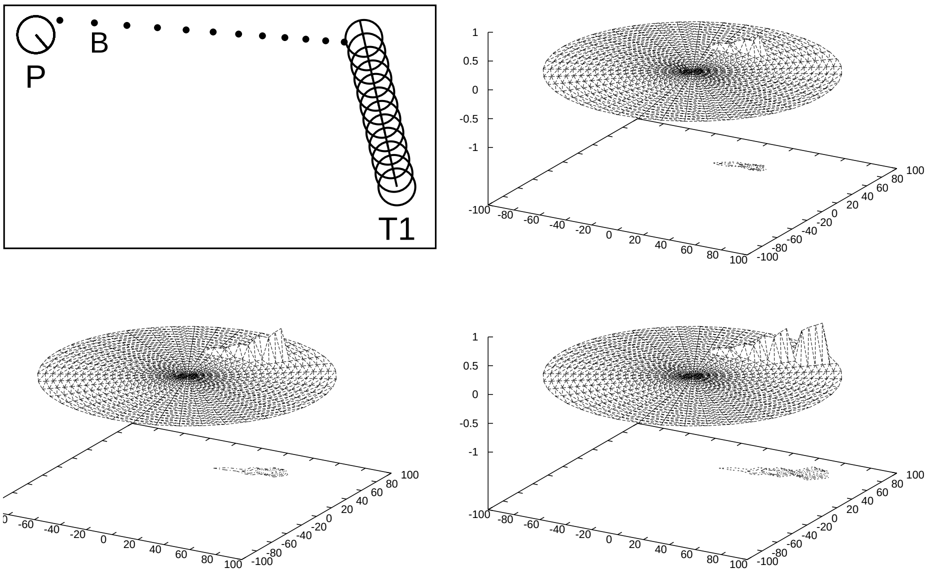


Fig. 56. Passing a ball to a moving team-mate utilizing the KicktoTeam map. Top left: the trace for the maximum speed on reaching its destination of three meters per second. Top right: the maximum speed is set to three meters per second. Bottom left: the maximum speed is set to five meters per second. Bottom right: the maximum speed is set to eight meters per second.

the impulse is calculated based on the momentum the ball needs to reach a location at the same time as the team-mate. Then, the impulses resulting in a too high speed are rejected. When the ball needs to travel to the collision point in a short time, the speed on reaching its destination is unavoidably high and thus the corresponding action is rejected.

Fig. 57 shows an example of a situation in which the player should score a goal. The KicktoGoal map at the top has a heavy weight for all maximum impulses directing the ball towards the goal. When the DontKicktoOpp maps are combined with the KicktoGoal map, only a single impulse is left in the MakeGoal map. If the opponents move closer to each other, the player has no possibility of scoring a goal. The DontKicktoOpp maps are difficult to calculate, as the impulses that should be avoided depend on the behavior of the opponents. In this experiment, the opponents were supposed to be quite effective in gaining possession of the ball and thus each blocked a wide sector.

In the second set of experiments, we tested the Samba for Soccer control system in the Soccer Server (Kitano *et al.* 1997). In these experiments, our team played against the teams developed by other researchers. Our team consists of 11 players, each connected to a separate Samba control system. Each Samba control system produced commands at the maximum frequency accepted by the Soccer Server, 10 Hz. Although the environment was simulated, maintaining this frequency was a strict requirement. This is because a team commanded at a slower rate gives too big an advantage to an opponent commanded at the maximum rate.

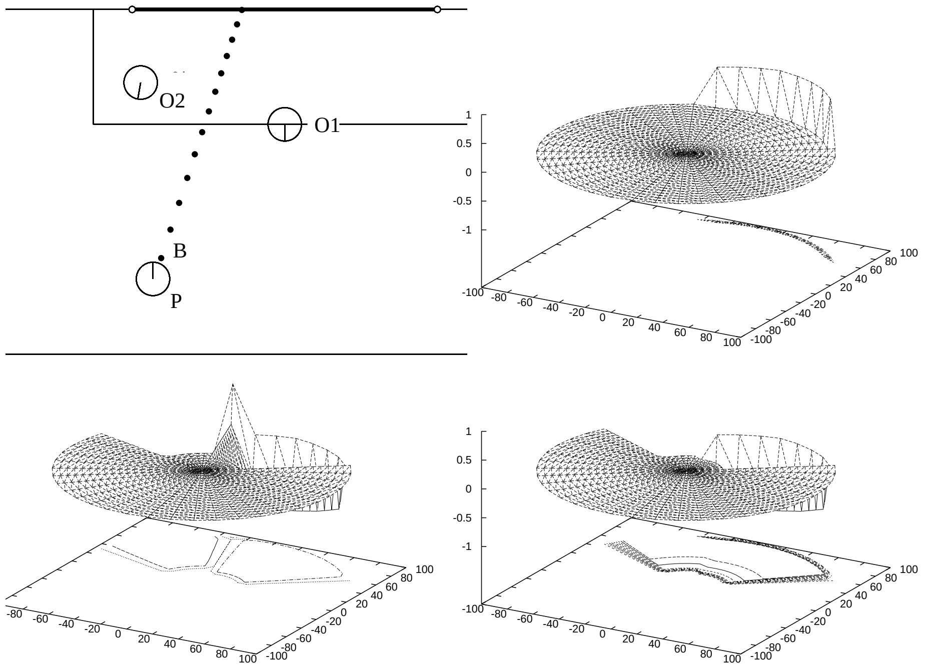


Fig. 57. The player (P) should score a goal. Top left: the trace. Top right: the KicktoGoal map. Bottom left: the MakeGoal composite map that resulted in the kick shown in the trace. Bottom right: the MakeGoal map for a situation in which the opponents (O1 and O2) have moved slightly closer to each other than is shown in the trace. The maps were calculated for a player facing to the right, although the player in the trace is facing the goal.

In addition to playing games against the publicly available teams, we carried out experiments at the RoboCup competitions. The first RoboCup competition was organized at Nagoya in August 1997. In the first round, we won two out of three matches. In the second round (elimination round) we lost. From 30 participants we qualified for the best 16. Fig. 58 shows a situation in our first match. Our team played from right to left. Player number five dribbles the ball near the opponents' goal. As the goal is blocked by opponents, it passes the ball to player number eight, who in turn passes the ball to player number 11. Next, player number eight circumvents the opponents. Finally, player number 11 passes the ball back to player number eight who scores a goal. In Fig. 58, player number eight has just passed the ball to player number 11.

The second RoboCup competition was organized at Paris in July 1998. We qualified at the competition for the quarterfinals from 32 participants. The competition, and preparation for it, are discussed in more detail in the next section.

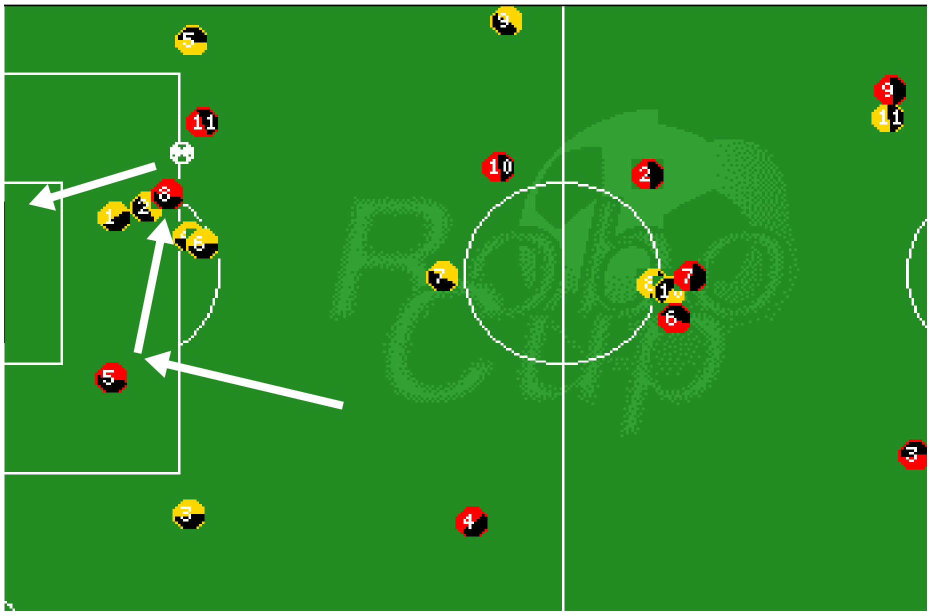


Fig. 58. Our team, CAT Finland, is attacking from right to left. The arrows show the trajectory of the ball. Players number 5, 8, and 11 attack and score a goal. The opponents block first player number 5 and then move towards player number 8.

5.3 Discussion

The experiments about action maps show that a Samba control system is able to understand environment dynamics. The player controlled by the Samba system reaches the right location at the right time. The system is able to maintain this property even when it executes several tasks in parallel. Further, a player reacts quickly to changes in the environment.

The Samba for Soccer system illustrates executing tasks in parallel. Even though the action with the heaviest weight in a composite map does not necessarily give maximum satisfaction to any one primitive behavior, it can still adequately satisfy several primitive behaviors. For example, the CatchBall behavior can select an action that does not reach the ball in the shortest possible time or avoid the opponents as much as possible but yet succeeds in both tasks.

When preparing for the first Robocup in Nagoya, most of our effort was spent on developing individual behaviors for independent players. It was a pleasant surprise that the players, although independent, did actually operate as a team, were able to dribble the ball despite the opponents, pass the ball on to team-mates, and once in a while score a goal. This first control system utilizing action maps proved that the calculation of action maps is computationally not as expensive as one might expect. The calculation of a primitive action map took 0.5-3 ms on a Sun Sparcstation 20.

During the competition, the pass play of our team worked well. Individual players could mostly select an optimal direction and a suitable team-mate to pass the ball to. This resulted in good and sometimes amazing pass combinations, where the ball was moved towards the opponents' goal efficiently. Unfortunately, our players were not aggressive enough when the ball was within scoring distance. Instead, they had a tendency to pass once too often, hence the scoring situation was lost. Our pass play operated well as long as the opponent was not able to move with greater speed than our players. The defence play was also quite efficient. The immediate front of our goal was kept clear (most of the time), but this resulted too often in a side kick. This was one of the weaknesses of our team. Furthermore, the primitive operation for taking a side kick was not good enough, causing failures and thus giving advantage to the opponent.

When preparing to the RoboCup'98, we optimized the team behavior by modifying the arbiter from the one described earlier. That is, we maximized the difference between the amount of goals scored by our team and the amount of goals scored by the opponent. We tested our team mainly against the champion of the RoboCup'97 World Soccer Championships, AT Humboldt. As this team plays a very effective game scoring at a high percentage, our team had to play a game with as few risks as possible. This resulted in a strategy where the players kicked the ball most of the time towards the opponents' goal as hard as possible, at the same time avoiding the opponents. This was accomplished with a composite map that was compiled from the KickFar map and the AvoidOpp maps. If our players started to dribble the ball, too often the opponents succeeded in stealing the ball, which gave them a further possibility of scoring. Further, if the players passed the ball to a team-mate, the ball did not move towards the opponents' goal as fast as when utilizing the KickFar map. Hence, the risk of an opponent scoring a goal was higher. On the other hand, when we practiced against teams that were not quite as effective as AT Humboldt, dribbling and passing the ball to team-mates resulted in more goals than always kicking the ball towards the opponents' goal. This was because the players lost the ball less often and the more controlled attacks resulted in better scoring situations.

After testing our team against AT Humboldt we started to play against the runner up of RoboCup'97, Andhill from Japan. These matches revealed interesting features. Although we had won most of the test matches against AT Humboldt and Andhill had lost in the final of RoboCup'97 against AT Humboldt, we lost our first test matches against Andhill. This was because Andhill plays a very different game when compared with AT Humboldt. The German team has players in good locations and moves the ball quickly across the field with strong kicks. The players form a chain towards the goal, each one kicking the ball to the next player and giving more speed to the ball. As a result, if the midfield is left too open, AT Humboldt develops a dangerous attack in a couple of seconds resulting in a very powerful shot at goal.

A key in beating AT Humboldt is to close the midfield so that these dangerous attacks are prevented. Andhill, on the other hand, uses the sides of the field much more effectively. When our team was tuned to block the middle field, Andhill was able to create a lot of attacks from the sides. These experiments illustrate the complexity of soccer as an application domain. Two teams can not be ordered based on games against a third team. The efficiency of a team depends so strongly on the opponent's ability that the result of a game cannot be predicted. We succeeded in improving our team play so that it also beat Andhill, but as a result the team scored less goals against AT Humboldt.

At the competition, we won four games and lost three. Our greatest defeat was the 0-8 defeat against the 1998 version of AT Humboldt. However, even in this game action maps worked well: the players moved and performed kicks with a skill nearly comparable with the opponent. But, our player's were not as good in dribbling the ball as the opponent. The passes performed by the opponent also found the right target more often than the passes performed by our team. Furthermore, our goalkeeper made some bad mistakes.

Soccer as an application has had a major impact on our research. The Soccer Server has facilitated in performing a large number of experiments in a highly dynamic environment. Soccer is also an interesting application, because it shows how difficult it is to decide on paper which situations are important in an application and how the robot should operate in those situations. For example, it seems to be important for a soccer player to avoid other players. Actually, the worst thing to do is to avoid an opponent approaching you. Furthermore, the skill of avoiding opponents has no noticeable correlation with the results. Due to modifications in the Soccer Server, we even ran some experiments in which the opponent could not see our team. This was not obvious when observing the game. The opposing team played quite an effective game when they were not aware of our players. Further, another important skill in soccer seems to be the ability to stay on the field. However, our team plays a better game when they do not consider the boundary lines at all. As all the other players and the ball are on the field, reactions to them tend to keep the reacting player on the field.

Maybe the most important lesson that we have learned from soccer is the insight about building situated robots. We have learned that it is not necessary to be able to find in the problem space the optimal path (or even any path at all) to the goal state from every possible state. Instead, it suffices to find a feasible action that decreases the distance to the goal state.

Action maps are one example: if the optimal action is prevented, an action guiding the robot closer to the goal location is selected. Another example is the behavior of our player when it is in a good location to score a goal, but the ball would bounce from the player itself if kicked towards the goal. One solution would be to calculate a sequence of kicks that would first move the ball around the player and then to the goal. Our player selects some other action in such a case. For example, if the player chooses to dribble the ball instead, it will probably be possible to score a goal a few moments later.

In other words, in a complex and dynamic environment the behavior satisfying the objectives of the robot can emerge from a carefully selected set of primitive tasks. In addition to being situated, the resulting system is also considerably simpler compared to one planning optimal actions. An example of an emerging behavior is an attack performed by our team: there is no mechanism directing the players towards the opponents' goal just because the team is attacking. Instead, the players favor the direction of the opponents' goal when kicking the ball. The location of the ball, in turn, changes the default locations of the players. As a result, when our team has the ball, they kick it towards the opponents' goal and move themselves to the same direction. In other words, they attack.

6. Discussion

The Samba architecture is built by extending a reactive, behavior-based architecture. The main extensions are markers and action maps. Markers are a representation for goals and intermediate results. Action maps are a representation for robot actions. These extensions improve the behavior coordination capabilities, enable several tasks to be executed in parallel, facilitate taking environment dynamics into account, foster managing the complexity of the system, and facilitate focusing on the important objects in the environment.

Behavior-based robots have been capable of executing only simple tasks, because the simple behavior arbitration method, the loss of information on the signal path, and the lack of representation of goals make it difficult to build versatile behavior coordination methods. In the Samba architecture, these problems are tackled by markers, action maps, and arbiters. Markers improve behavior coordination. First, markers activate the behaviors needed to perform the task and deactivate the behaviors preventing the task. Second, markers share data among the behaviors and focus their attention on the task-related environment objects.

In traditional behavior-based architecture, the arbitration is based on static, predefined priorities. In the Samba architecture, the priorities are dynamic. They are calculated according to the environment state and the task being executed. Further, as an action map describes preferences for all possible different actions, information is not lost on the signal path. This facilitates executing several tasks in parallel. Behavior has seldom such strict conditions that only one action suffices. Rather, different actions can be weighted – there is an action that gives the maximum satisfaction to the behavior, but there are also many other actions that are good enough. By utilizing an action map, a behavior can give heavy weights for all actions that are good enough. When several such action maps are combined, the best action in the resulting map does not necessarily give maximum satisfaction to any one behavior, but still satisfies several behaviors.

The capability to execute several tasks in parallel leads directly to reactive task execution. A robot capable of executing tasks in parallel can both execute the reasoned task and react to unexpected events at the same time. A further requirement for reactive task execution is that reactions must be fast. The Samba architecture fulfills this requirement, as the signal path of the primitive reactions is short. As a behavior-based system, a Samba control system reacts quickly to environment events. The new features do not destroy the reactiveness, because they do not extend the signal paths, increase the amount of computation, nor increase the amount of stored information considerably.

The method for combining commands (actions) has the important advantage that an executed command is always proposed by at least one behavior. When vector summation is used for behavior coordination, there is no guarantee that the resulting behavior is reasonable. In the Samba architecture this problem is avoided.

Action maps facilitate taking environment dynamics into account. A primitive action map contains high weights for actions that cause an object to be reached, even if the object is moving. By combining several such maps a composite behavior can, for example, avoid a moving obstacle while reaching a moving target.

Markers and action maps also foster managing the complexity of the system. Markers are utilized in decomposing the system into subsystems, both into goal-achieving subsystems and into producers and consumers of data. Further, action maps are used in decomposing the system into producers and consumers of primitive reactions. Finally, task markers serve as an interface between the Samba control system and the task layer.

The work on the Samba architecture has not yet been completed. Action maps need to be tested on real robots. To do that, sensor modules capable of binding and updating the object markers are needed. The task markers should also be tested more extensively. This would require a more complex task layer. Furthermore, the methods for calculating action maps should be developed further. In this work, the emphasis was on the OSA method because reaching the target in the shortest possible time is the most important requirement for a robot playing soccer. When the requirements for the robot speed are not so strict the MSA method produces better results. For such applications, the MSA method could be improved to ground all weights in the environment. Furthermore, new procedures for calculating action maps could be developed to control a mobile robot in narrow indoor environments.

Although more work needs to be done on the Samba architecture, the work done so far has already verified that the architecture facilitates reactive task execution. The experiments on the Pure control system verified the hypotheses that markers improve a behavior-based system by increasing the modularity of the control system, which assists in extending the reactive control systems with goal-oriented capabilities. A further advantage was that conditions for tasks were easy to express using markers and their spatial relations. Finally, all these improvements were achieved without sacrificing the real time performance of a behavior-based system, as the Pure control system was successfully executed at frequencies over 20 Hz.

The experiments on the Pure control system taught us the importance of grounding everything physically. Updating even the simple representations, markers, turned out to be difficult. To overcome this problem, we suggest that a stored representation of the environment should be interpreted as directions for perceiving the corresponding values directly from the environment. In this approach, the representation indexes an object in the environment. When a task is being performed, the representation is used to find the object and the accurate values are perceived. Markers can be used to function as such an indexing representation when each marker associates a perceivable environment feature with task related data.

The experiments on the Samba for Soccer control system confirmed that when actions are represented as action maps, tasks can be executed in parallel. Furthermore, the system took environment dynamics into account. A player controlled by the Samba system reached the right location at the right time, even when the correct action required several moving objects to be considered. Also the action maps preserved the reactive properties of the system, as the Samba control system was successfully executed at a frequency of 10 Hz.

The experiments on the Samba control system also gave us insight into building situated robots. We learned that it is not necessary to be able to find in the problem space the optimal path (or even any path at all) to the goal state from every possible state. Instead, it suffices to find a feasible action that decreases the distance to the goal state. In other words, as the reactive, local approach of generating actions directly based on sensor data produced such good results, there was no need for a more exhaustive, global approach. These insights, the importance of grounding everything physically and the efficiency of the reactive approach of executing tasks confirm the importance of situatedness advocated by Brooks (1991a).

7. Conclusions

In this work, we suggested reactive task execution to be a central requirement for a robot operating in our everyday environment. We defined reactive task execution as integrating task execution with fast reactions to unexpected situation. The capability to execute tasks in parallel, also in a dynamic environment, was specified as essential in reactive task execution.

We suggested a novel control architecture, Samba, for reactive task execution. The requirements set by reactive task execution are fulfilled by the representations of goals, intermediate results, and robot actions. Markers describe goals and intermediate results. They facilitate building versatile behavior coordination methods and managing the system´s complexity. Markers decompose the system vertically into producers and consumers of data. Furthermore, they form a horizontal interface to a higher layer system.

The key idea in Samba is to produce continuously primitive reactions for all the important objects in the environment. These reactions are represented as action maps. Tasks are executed by modifying and combining the action maps. The tasks can be either reasoned by a higher layer or triggered by sensor data. As the reactions are produced continuously based on sensor data, the robot actions are based on the current state of the environment. Action maps enable executing tasks in parallel and considering environment dynamics.

We tested markers on an actual robot equipped with a stereo gaze platform. Further, we applied the Samba architecture to playing soccer and carried out experiments in the 1997 and 1998 RoboCup competitions. In addition to validating the Samba control architecture, the experiments confirmed the importance of situatedness. First, they taught us the utmost importance of grounding everything physically. Second, the experiments suggest that planning optimal action sequences is rarely necessary as generating plausible actions directly based on sensor data produces such good results.

The experiments validated that markers and action maps facilitate fulfilling the requirements for reactive task execution. Furthermore, the experiments verified that markers and action maps do not destroy the reactive properties of the underlying behavior-based architecture. From these results we can conclude that the Samba architecture is a potential alternative for mobile robots operating in dynamic environments.

References

Agre PE & Chapman D (1990) What are plans for? In: Maes (ed) Designing Autonomous Agents: Theory and Practice from Biology to Engineering and Back. MIT Press, Cambridge, MA, p 17-34.

Agre PE & Chapman D (1995) Pengi: An implementation of a theory of activity. In: Luger (ed) Computation & Intelligence: Collected Readings. MIT Press, Cambridge, MA, p 635-644.

Arkin RC (1990) Integrating behavioral, perceptual, and world knowledge in reactive navigation. In: Maes (ed) Designing Autonomous Agents: Theory and Practice from Biology to Engineering and Back. MIT Press, Cambridge, MA, p 105-122.

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

Brooks RA (1986b) Achieving artificial intelligence through building robots. MIT AI Memo 899. May 1986.

Brooks RA (1987) A hardware retargetable distributed layered architecture for mobile robot control. Proc IEEE International Conference on Robotics and Automation, Raleigh, North Carolina, p 106-110.

Brooks RA (1989a) A robot that walks; emergent behaviors from a carefully evolved network. MIT AI Memo 1091. February 1989.

Brooks RA & Flynn AM (1989b) Fast, cheap and out of control: a robot invasion of the solar system. Journal of the British Interplanetary Society (42): 478-485.

Brooks RA (1990) The behavior language; users guide. MIT AI Memo 1227. April 1990.

Brooks RA (1991a) New approaches to robotics. Science (253): 1227-1232.

Brooks RA (1991b) Intelligence without reason. MIT AI Memo 1293. April 1991.

Brooks RA (1991c) Intelligence without representation. Artificial Intelligence (47): 139-159.

Chapman D (1987) Planning for conjunctive goals. Artificial Intelligence 32(3): 333-377.

Chapman D (1989) Penguins can make cake. AI Magazine 10(4):45-50.

Chapman D (1991) Vision, instruction, and action. MIT Press, Cambridge, MA.

Connell JH (1987) Creature design with the subsumption architecture. Proc Tenth International Joint Conference on Artificial Intelligence, August 1987, pp. 1124-1126.

Ferguson IA (1994) Autonomous agent control: A case for integrating models and behaviors. Proc AAAI Fall Symposium on Control of the Physical World by Intelligent Agents, November 1994, pp 46-54.

Ferrell CL (1995) Global behavior via cooperative local control. Autonomous Robots (2):105-125.

Gat E (1992) Integrating planning and reacting in a heterogeneous asynchronous architecture for controlling real-world mobile robots. Proc National Conference on Artificial Intelligence (AAAI), July 1992, pp 809-815.

Gat E (1993) On the role of stored internal state in the control of autonomous mobile robots. AI Magazine 14(1):64-73.

Gini M (1988) Integrating planning and execution for sensor-based robots. In: Ravani (ed) CAD Based Programming for Sensory Robots. NATO ASI Series vol. F50. Springer-Verlag, Berlin, p 255-274.

Ginsberg ML (1989a) Universal planning: an (almost) universally bad idea. AI Magazine 10(4): 40-44.

Ginsberg ML (1989b) Universal planning research: a good or bad idea? AI Magazine 10(4): 61-62.

Kitano, H., Asada, M., Kuniyoshi, Y., Noda, I., Osawa, E. & Matsubara H. (1997) RoboCup: A challenge program for AI. AI Magazine, Spring 1997: 73-85

Kaelbling LP & Rosenschein SJ (1990) Action and planning in embedded agents. In: Maes (ed) Designing Autonomous Agents: Theory and Practice from Biology to Engineering and Back. MIT Press, Cambridge, MA, p 35-48.

Kuniyoshi, Y., Riekki, J., Ishii, M., Rougeaux, S., Kita, N., Sakane, S. & Kakikura, M. (1994) Vision-Based Behaviors for Multi-Robot Cooperation. Proc IEEE/RSJ/GI International Conference on Intelligent Robots and Systems (IROS'94), Munchen, Germany, September 1994, pp 925-932.

Malcolm C & Smithers T (1990) Symbol grounding via a hybrid architecture in an autonomous assembly system. In: Maes (ed) Designing Autonomous Agents: Theory and Practice from Biology to Engineering and Back. MIT Press, Cambridge, MA, p 123-144.

Mataric MJ (1997) Behavior-based control: Examples from navigation, learning, and group behavior. Journal of Experimental and Theoretical Artificial Intelligence 9(2-3):323-336.

Oxford dictionary of current English (1985) Oxford University Press.

Parker LE (1994) ALLIANCE: An architecture for fault tolerant, cooperative control of heterogeneous mobile robots. Proc IEEE/RSJ/GI International Conference on Intelligent Robots and Systems (IROS'94), Munchen, Germany, September 1994, pp 776-783.

Payton DW (1986) An architecture for reflexive autonomous vehicle control. Proc 1986 IEEE International Conference on Robotics and Automation, San Francisco, USA, p 1838-1845.

Payton DW, Rosenblatt JK & Keirsey DM (1990) Plan guided reaction. IEEE Transactions on systems, man, and cybernetics 20(6): 1370-1382.

Rosenblatt JK & Payton DW (1989) A fine-grained alternative to the subsumption architecture for mobile robot control. Proc IEEE/INNS International Joint Conference on Neural Networks, Washington DC, 2: 317-324.

Rosenblatt JK & Thorpe CE (1995) Combining multiple goals in a behavior-based architecture. Proc IEEE/RSJ International Conference on Intelligent Robots and Systems, Pittsburgh, USA, p 136-141.

Rosenblatt JK (1997) DAMN: A distributed architecture for mobile navigation. PhD thesis, Carnegie Mellon University, USA.

Rougeaux, S., Kita, N., Kuniyoshi, Y., Sakane, S. & Chavand, F. (1994) Binocular tracking based on virtual horopters. Proc IEEE/RSJ/GI International Conference on Intelligent Robots and Systems (IROS'94), Munchen, Germany, September 1994, pp 2052-2058.

Schoppers MJ (1987) Universal plans for reactive robots in unpredictable domains. Proc. Tenth International Joint Conference on Artificial Intelligence, Milan, Italy, p 1039-1046.

Schoppers M J (1989) In defense of reaction plans as caches. AI Magazine 10(4): 51-60.

Tsotsos JK (1995) Behaviorist intelligence and the scaling problem. Artificial Intelligence 75: 135-160.

Webster's Encyclopedic Unabridged Dictionary of the English Language (1996) Random House.

Weld DS (1994) An introduction to least commitment planning. AI magazine 15(4): 27-61.

Winston, PH (1992) Artificial Intelligence, 3rd ed., Addison-Wesley, Reading, Massachusetts.

Zelinsky A & Kuniyoshi Y (1996) Learning to coordinate behaviours in mobile robots. Journal of Advanced Robotics 10(2):143-159.