Collaborative Robot Agents

THESIS

submitted in partial fulfillment of the
requirements for the degree of

MASTER OF SCIENCE

in

DATA AND KNOWLEDGE SYSTEMS

by

E. Daman BSc.
The Netherlands

This work was performed for:

Man-Machine Interaction Group
Department of Mediamatics
Faculty of Electrical Engineering, Mathematics and Computer Science
Delft University of Technology

Collaborative Robot Agents

by E. Daman BSc., 1014900

Copyright © 2008

Abstract

This thesis investigates how methods for collaborative agents can be applied to the AIBO (*A*rtificial *I*ntelligence ro*BO*t) of SONY Corporation. A method for collaboration will be explained in detail, and the design and implementation of a working prototype will be presented. The software works in an ad-hoc wifi network with clients that can connect and disconnect at all times. Data can be shared between the clients (AIBOs) so that they can determine the right objectives for themselves and fine-tune their actions with the others.

Group	:	Man-Machine Interaction Group
Faculty	:	Faculty of Electrical Engineering, Mathematics and Computer Science
Institute	:	Delft University of Technology

Advisor:

prof.drs.dr. L.J.M. Rothkrantz, MM-MMI, TU Delft

Committee Member(s)

dr. K. van der Meer, ST-SE, TU Delft

dr.ir. C.A.P.G. van der Mast, MM-MMI, TU Delft

ir. Z. Yang, MM-MMI, TU Delft

Contents

List of Figures

List of Tables

Preface

In the course of studying Computer Science at Delft University of Technology (TUD), I conducted a research in the area of Collaborative Robot Agents. The research is focused on the AIBO of SONY Corporation. It was done in the period of March 2005 to September 2008. This thesis report contains the results of the research.

I am deeply grateful to my advisor prof.drs.dr. Rothkrantz, who gave me much help during my research with high patience and enthusiasm, even during the entire timespan that writing this report took. I also would like to thank Zhenke Yang for helping me out with all the technical aspects of AIBO, and for his valuable comments on my thesis report. And finally, many thanks go to my parents, for their unfailing support over the years. Without help and support of these people, this thesis would never have been written.

Evert Daman, 1014900
Delft, The Netherlands
September, 2008

Glossary

Definitions

- **Situation**
  - complete state of the world at a particular point in time
  - might include agents' beliefs (incomplete knowledge)
- **Action**
  - function from one situation to another
  - changes situation to another situation, given preconditions
- **Recipe (Plan)**
  - sequence of actions to perform
- **Objective (Goal)**
  - an intention driving current behavior
  - actions defined in terms of their effects
- **Resource**
  - an object used during execution of a recipe
  - consumable or reusable.

Introduction 1

In computer science, a software agent is a piece of software that acts for an user or other program in a relationship of agency. Such 'acting on behalf of' implies the authority to decide which (and if) action is appropriate. The idea is that agents are not strictly invoked for a task, but activate themselves.

To date, agents have been successfully employed in multiple application endeavors, such as:

- data collection and filtering
- pattern recognition
- event notification
- data presentation
- planning and optimization
- rapid response implementation.

One might imagine an agent which is involved in all of these activities, sequencing through each of those states in response to environmental events, acting independently or in collaboration with a human client.

The number of application areas for software agents continues to expand. Currently, software agents are primarily used in research, but other areas such as education, commercial, law enforcement, intelligence, and even military are showing their interest as well, as future clients increasingly find themselves swamped by information overload. Agents can process vast volumes of data which would be unmanageable by human agents. Additionally, agents can analyze, assess, plan, and react to environmental events at super-human speeds.

A problem with designing agents is the fact that their competence is necessarily restricted to situations similar to those they have encountered in the past. The agents, when working on their own, have only a limited vision of the world, and have their resource limitations. If social ability is introduced however, and multiple agents are able to engage other components through some sort of communication and coordination, they may collaborate on a task, and collect information that would not be available to them if they were working alone. They could share knowledge and resources, which could lead to better decisions based on more information, and tasks may be completed in less time.

As with every advantage, sharing knowledge has its problems as well. The agents need to deal with issues like distributed or centralized data, the storage of data, and have to deal with multiple sources of information. Received and stored information may not be up-to-date anymore, and the original source of that information may be disconnected from the network of agents. Tradeoffs have to be made: is information distributed between all the agents, even to agents that can't see the original source? Maybe a centralized control center which keeps track of all the information and assigns tasks to the agents is more applicable? What to do with history of data, and do we trust the information that is received from other agents?

1.1 The Collaboration Problem

When several intelligent agents interact they may form a multi-agent system or multiple-agent system. The agents are considered to be autonomous entities collaborating with each other, and their interactions can be either cooperative or selfish. That is, the agents can share a common goal (e.g. an ant colony), or they can pursue their own interests (as in the free market economy).

Multi-agent systems can be used to solve problems which are difficult or impossible for an individual agent or monolithic system to solve. Examples of problems which are appropriate to multi-agent systems research include online trading, disaster response, and modeling social structures.[1] They have been successfully applied in domain areas like entertainment, computer games, air-traffic control, air combat, personal assistants, load-balancing, logistics, mining large (distributed) datasets in astronomy or transportation, finance, geographic information systems as well as in many other fields. Multi-agent systems are widely being advocated to be used in networking and mobile technologies, to achieve automatic and dynamic load balancing, high scalability, and self healing networks.

Central to the design and effective operation of collaborative agents are a core set of issues and research questions that have been studied over the years by the distributed AI community:[2]

- How do we formulate, describe, decompose, and allocate problems and synthesize results among a group of intelligent agents?
- How do we enable agents to communicate and interact? What communication languages and protocols do we use? What and when can they communicate? How can we find useful agents in an open environment?
- How do we ensure that agents act coherently in making decisions or taking action, accommodating the nonlocal effects of local decisions and avoiding harmful interactions? How do we ensure the agents do not become resource bounded? How do we avoid unstable system behavior?
- How do we enable individual agents to represent and reason about the actions, plans, and knowledge of other agents to coordinate with them. How do we reason about the state of their coordinated process (for example, initiation and completion)?

©2008 Delft University of Technology

- How do we recognize and reconcile disparate viewpoints and conflicting intentions among a collection of agents trying to coordinate their actions?
- How do we engineer and constrain practical distributed AI systems? How do we design technology platforms and development methodologies for multi-agent systems?

Typically multi-agent systems research refers to software agents. However, the agents in a multi-agent system could equally well be robots, humans or human teams. A multi-agent system may even contain combined human-agent teams. This thesis primarily focuses on robot agents, which are used in various kinds of applications:

- robots as entertainment
- robots as watchdogs
- security and surveillance
- robots as social entities.

Each of these applications differ in the kind of information that they gather, and the actions each of the agents can perform. Since the information they use differs, the way they could share it differs as well. A robot that is used for entertainment may collect information such as 'where can I find the charging station' or 'where are my toys', while a robot in a search-and-rescue situation collects information about missing people or 'where is the location of the fire'. Sharing information and keeping it up-to-date in the last example is of greater importance then in the entertainment example, because in a search-and-rescue situation every second counts, and having the correct information is much more valuable. Not having agents/robots explore rooms that have already been searched by distributing knowledge between them, can significantly reduce the time of the rescue mission.

While the information that is shared and the way agents collaborate differ in each situation, the underlying technique of 'message passing' and 'communication' applies to all applications. A common communication protocol is needed that allows the agents to communicate and understand each other and to share information/knowledge with, and an expert system is needed with which the agents can make decisions based on all the information that they gather from other agents. Furthermore, choices have to be made whether to keep the agents/robots autonomous, or use a central controller PC, and whether to save and use historic data and how to deal with 'trust'. All of these decisions should be based on the demands that each different application described above has. When all this is carefully chosen, designed and implemented, collaboration could give a huge benefit over single operating agents.

1.2 Research Objectives

The research objective described in this thesis is the design and development of a system of collaborative agents, applied to AIBO. Collaboration methods must be investigated in

©2008 Delft University of Technology

detail and a protocol that can be used on the AIBO should be developed. The software should work in an ad-hoc wifi network with clients that can connect and disconnect at all times. Data can then be shared between the clients (AIBOs) so that they can determine the right objectives for themselves and fine-tune their actions with the others.

The application that this thesis primarily focuses on is the 'Four-Legged Robot Soccer League'. Therefore the robots should stay autonomous, since the rules of the RoboCup soccer game clearly states that the use of a central controller PC is forbidden.

Figure 1.1: Four-Legged Robot Soccer League

The followings tasks contribute to achieving the research objective:

- Understanding of current collaborating techniques, by studying relevant literature.
- Evaluation of the different development platforms that are available for AIBO.
- Study of the expert systems that are currently available.
- Design of the software and corresponding communication protocol.
- Implementation of the software in a development platform of choice.
- Implementation of knowledge in an expert system.
- Evaluation of the software when running on AIBO.

The software that is going to be developed is called AiboCommOpenR.

1.3 About This Document

This document represents the partial fulfillment of the requirements for obtaining a Master of Science degree. Each of the research tasks listed above will be discussed in detail. In chapter 2 the different kind of agents are described and the collaborative agent is discussed in further detail. It also includes a detailed problem solving model, and

©2008 Delft University of Technology

discusses collaborative frameworks such as JADE. Chapter 3 is the result of studying the relevant literature related to communication protocols. In this chapter different protocols are described and a preliminary design of the protocol for AiboCommOpenR is given. Chapter 4 gives an overview of all the available development platforms for AIBO, and chapter 5 discusses the development platform used for this project, called OPEN-R SDK, in more detail.

Chapter 6 describes the design of AiboCommOpenR including the needed communication protocol, and discusses some design tradeoffs. In Chapter 7 a few possible expert systems are introduced including their arguments for and against. It results in an expert system that can be used in this project. Chapter 8 describes some applications of the technique and introduces a few strategies for the Four-Legged Robot Soccer League, which is the application that is chosen in chapter 9 to design expert system rules for.

Finally, in chapter 10 AiboCommOpenR is put to the test, and chapter 11 concludes this thesis and indicates directions for future work.

©2008 Delft University of Technology

©2008 Delft University of Technology

2 Collaborative Robot Agents

A global definition of the term ‘agent’ would be ‘a component of software and/or hardware which is perceiving its environment through sensors and acting on the environment through actuators, thus effecting what it senses in the future’.[3] The term percept is used to refer to the agent’s perceptual inputs at any given instant. Perception is initiated by sensors and provides agents with information about the world they inhabit (see figure 2.1).

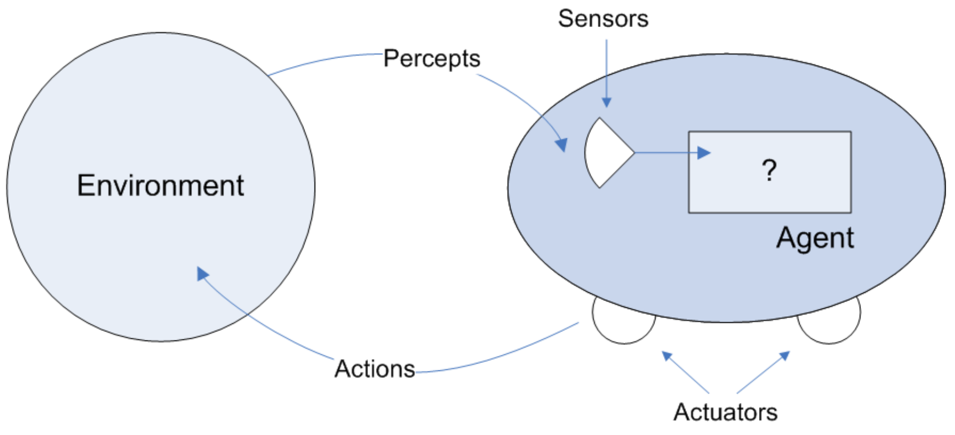


Figure 2.1: Agent definition

This tends to obscure the differences between radically different approaches. Agents differ in the way they communicate and co-operate and so on. The different kind of agents can be split into 7 categories:[4][5]

- Collaborative agents
- Interface agents
- Mobile agents *(static vs. mobile)*
- Information/Internet agents
- Reactive agents *(deliberative vs. reactive)*
- Hybrid agents
- Smart agents.

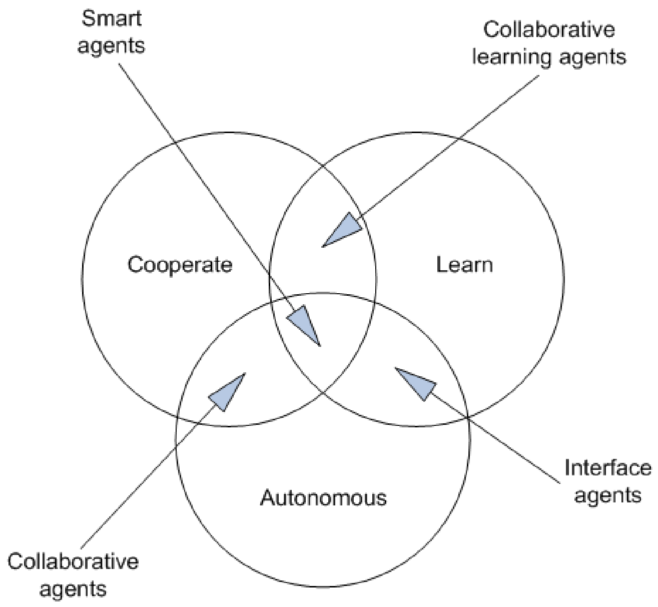


Figure 2.2: Agent classification

All these type of agents have different motivations and challenges. This thesis investigates the 'collaborative agents'. These agents negotiate in order to resolve conflicts and they collaborate to integrate information. Collaborative agents can wrap around legacy systems ('glue' to interconnect them) and can provide solutions to inherently distributed problems like air traffic control, telecommunications and network management.

An other classification scheme for agents is the weak and strong notion of agency. In the weak notion of agency, agents have their own will and operate without the direct intervention of humans or others (autonomy). They are able to interact with each other (social ability) and perceive their environment and respond to changes in it (reactivity). Finally they are able to exhibit goal-directed behavior by taking the initiative (proactiveness).

In the strong notion of agency (used particularly by researchers working in AI), the weak notions of agency are preserved and in addition agents can move around (mobility), they are truthful (veracity) and they do what they're told to do (benevolence). They will also perform in an optimal manner to achieve goals (rationality). Some AI researchers have gone further and considered emotional agents.[6]

2.1 Designing Collaborative Agents

A problem with designing agents is the fact that their competence is necessarily restricted to situations similar to those they have encountered in the past. A collaborative framework can alleviate these problems. When faced with an unfamiliar situation, an agent can consult its peers who may have the necessary experience to help the agent.

Collaborative agents need to be able to:[7]

- discuss and negotiate goals
- discuss options and decide on courses of action, including assigning different parts of a task to different agents
- discuss limitations and problems with the current course of action and negotiate modifications
- assess the current situation and explore future possibilities
- discuss and determine resource allocation
- discuss and negotiate initiative in the interactions
- perform parts of the task and report to others to update shared knowledge of the situation.

The main motivation factor for designing cooperative agents is the fact that some problems are simply to large for a centralized single agent. Resources might be limited or there is too much risk involved. But beside resources limitations, other motivations play an important role as well:[8]

- Allow for interconnection/interoperability of multiple existing legacy systems.
- Provide solutions to inherently distributed problems, e.g. distributed sensor networks or air-traffic control.
- Provide solutions which draw from distributed information sources.
- Provide solutions where the expertise is distributed, e.g. in health care provisioning.
- Enhance modularity (which reduces complexity), speed (due to parallelism), reliability (due to redundancy), flexibility (i.e. new tasks are composed more easily from the more modular organisation) and reusability at the knowledge level (hence shareability of resources).
- Research into other issues, e.g. understanding interactions among human societies.

So by letting agents cooperate we are able to:

> "create a system that interconnects separately developed collaborative agents, thus enabling the ensemble to function beyond the capabilities of any of its members"[9]

©2008 Delft University of Technology

The central issue of collaborative problem solving is that in order to collaborate to achieve a common goal, the agents must coordinate their individual actions. Therefore, some sort of 'Collaborative Problem Solving Level' must exist, and the agents should capture the joint objectives, and the jointly chosen recipes and resources etc. to accomplish those objectives.

The collaborative problem solving level must serve 2 purposes. It must provide the structure that enables and drives interactions, and it should provide the connection between the joint intentions and the individual actions that an agent performs as part of the joint plan, while allowing an agent to have its own objectives. Agents must agree to some level of detail on the abstract joint recipes that they construct in order to collaborate. An agent does not need to know how the other agent accomplishes things below this level of abstraction and agents do not need to have the same library of recipes.[5]

Being able to solve problems, the cognitive state of the agent, affected by problem-solving actions, need to contain at least the following:[7]

- current situation
- intended objectives
- active objectives
- intended recipes
- recipe library
- function on objectives and situations giving recipes.

As a summary, the different phases to agent problem solving can be summed as follows:[7]

1. Determine the objective *(agents may reconsider objectives at any time)*.
2. Determine the recipe *(find a recipe that may achieve the objective)*.
3. Use the selected recipe and identify the next action to perform from the recipe *(if the recipe returns a sub-objective, go to step 1)*.

2.2 Collaborative Framework

While a particular agent may not have any prior knowledge about a certain problem, there may exist a number of agents who do. Instead of each agent re-learning what other agents have already learned through experience, agents can simply ask for help. This gives each agent access to a potentially vast body of experience that already exists. There are two classes of such collaboration:[8]

- *Desperation based communication* which is invoked when a particular agent has insufficient experience to make a confident decision.

- *Exploratory communication* communication, on the other hand, is initiated by agents in bids to find the best set of peer agents to ask for help in certain classes of situations. Exploratory communication is undertaken by agents to discover new (as yet untried) agents who are better advisors in the solving of the agents' problems than the current set of peers they have previously tested.

Both forms of communication may occur at two orthogonal levels. At the *situation level*, desperation communication refers to an agent asking its peers for help in dealing with a new situation, while exploratory communication refers to an agent asking previously untested peers for how they would deal with old situations for which it knows the correct action, to determine whether these new agents are good advisors for future problems. At the *agent level*, desperation communication refers to an agent asking trusted peers to recommend an agent that its peers trust, while exploratory communication refers to agents asking peers for their evaluation of a particular agent perhaps to see how well these peers' modeling of a particular agent corresponds with their own.

Collaborative communication between agents occurs in the form of request and reply messages. An agent is not required to reply to any message it receives. This leaves each agent the freedom to decide when and whom to help.

Agents model peers' abilities to assist with solving problems by a trust value. For each class of situations an agent has a list of peers with associated trust values. Trust values vary between 0 and 1. The trust values reflect the degree to which an agent is willing to trust a peer's solution for a particular situation class. A trust value represents a probability that a peer's solution for a certain problem is a good solution based on prior history of proposed solutions from the peer. Trust values are further discussed in chapter 6.2.3.[5]

When an agent sends out a solution request to more than one peer its likely to receive many replies, each with a potentially different solution and confidence value. In addition, the agent has a trust value associated with each peer. This gives rise to many possible strategies which an agent can use to choose a solution and a confidence value for this solution. Both trust and peer confidence should play a role in determining which proposed solution gets selected and with what confidence. Each proposed action is assigned a trust-confidence sum value which is the trust weighted sum of the confidence values of all the peers predicting this action. The action with the highest trust-confidence sum is chosen.

2.3 A Detailed Problem Solving Model

The following problem solving model is primarily focused on human-machine collaboration, but can be equally well applied to interactions between sophisticated software agents that need to coordinate their activities.[7]

An overview of the model is shown in figure 2.4. At the heart of the model is the *problem solving level*, which describes how a single agent solves problems. An agent

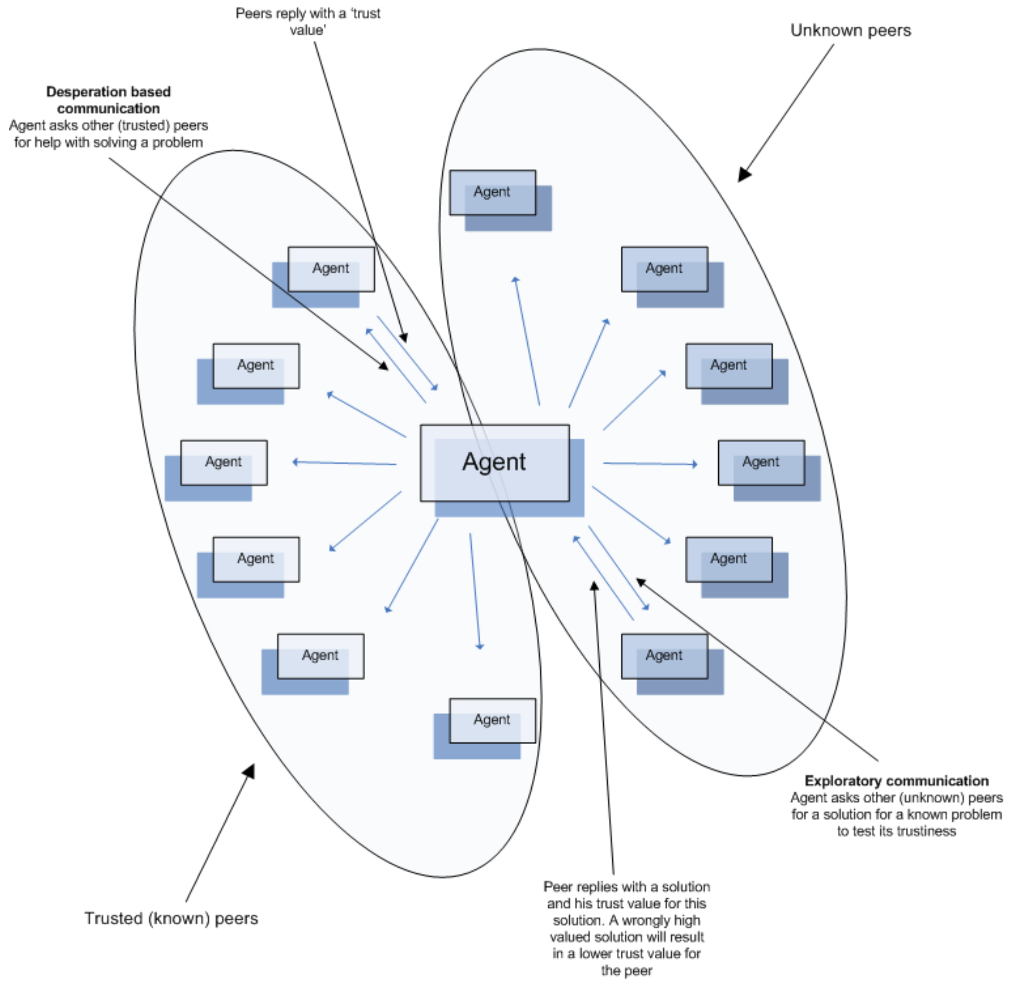

Peers reply with a 'trust value'
Unknown peers
Desperation based communication
Agent asks other (trusted) peers for help with solving a problem
Agent
Exploratory communication
Agent asks other (unknown) peers for a solution for a known problem to test its trustiness
Trusted (known) peers
Peer replies with a solution and his trust value for this solution. A wrongly high valued solution will result in a lower trust value for the peer

Figure 2.3: Communicating agents

might adopt an obligation or might evaluate the likelihood that a certain action will achieve that objective. This level is based on a fairly standard model of agent behavior. The problem solving level is specialized to a particular task and domain by a *task model*. The task model describes how to perform certain tasks, such as what possible objectives are, how objectives are (or might be) related, what resources are available, and how to perform specific problem solving actions such as evaluating a course of action. For an isolated autonomous agent, these two levels suffice to describe its behavior, including the planning and execution of task-level actions. For collaborative activity, more is needed.

The *collaborative problem solving level* builds on the single-agent problem solving level. The collaborative problem solving actions parallel the single-agent ones, except that they are joint actions involving jointly understood objects. For example, the agents can jointly adopt an intention (making it a joint intention), or they can jointly identify a relevant resource, and so on. Finally, an agent cannot simply perform a collaborative action by itself. The *interaction level* consists of actions performed by individuals in

©2008 Delft University of Technology

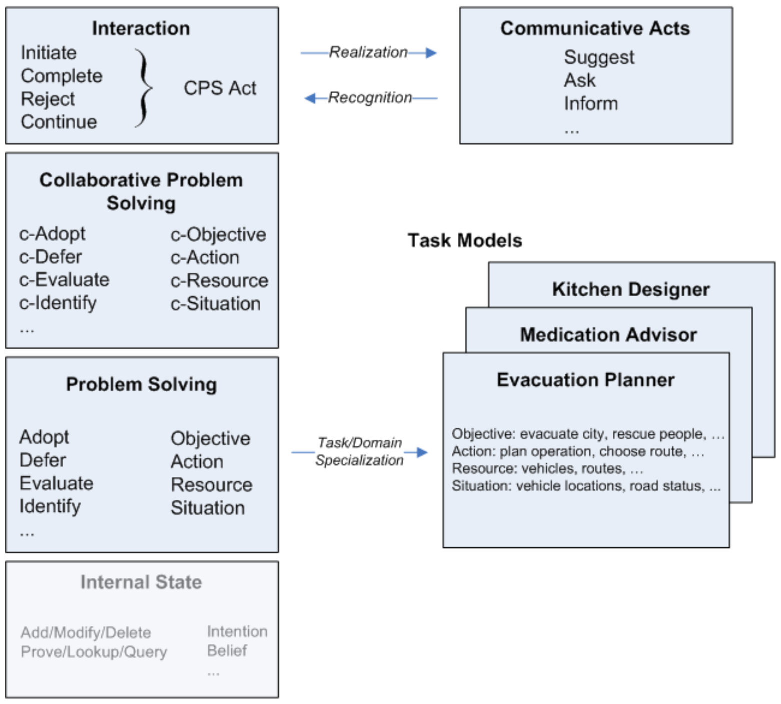


Figure 2.4: Collaborative problem solving model

order to perform their part of collaborative problem solving acts. Thus, for example, one agent may initiate a collaborative act to adopt a joint intention, and another may complete the collaborative act by agreeing to adopt the intention.

When two agents collaborate to achieve goals, they must coordinate their individual actions. To mirror the development at the problem solving level, the collaborative problem solving level operates on the collaborative problem solving (CPS) state, which captures the joint objectives, the recipes jointly chosen to achieve those objectives, the resources jointly chosen for the recipes, and so on.[7]

The collaborative problem solving model must serve two critical purposes. First it must provide the structure that enables and drives the interactions between the agents as they decide on joint objectives, actions and behavior. In doing so, it provides the framework for intention recognition, and it provides the constraints that force agents to interact in ways that maintain the collaborative problem solving state. Second, it must provide the connection between the joint intentions and the individual actions that an agent performs as part of the joint plan, while still allowing an agent to have other individual objectives of its own.

Establishing part of the collaborative problem solving state requires an agreement between the agents. One agent will propose an objective, recipe, or resource, and the other can accept, reject or produce a counterproposal or request further information.

©2008 Delft University of Technology

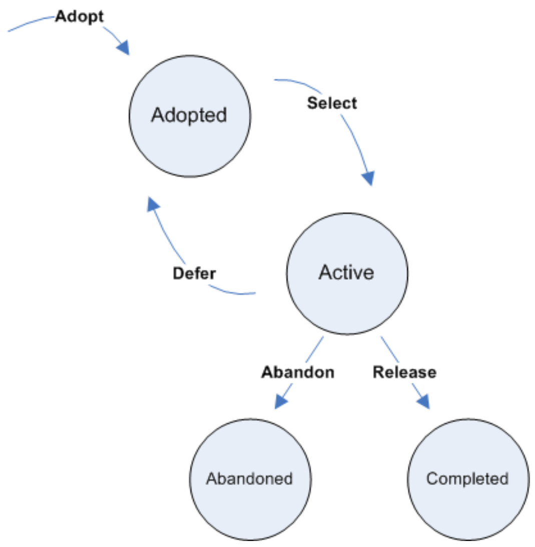


Figure 2.5: Life cycle of an intention

This is the level that captures the agent interactions. To communicate, the agent receiving a message must be able to identify what CPS act was intended, and then generates responses that are appropriate to that intention.

2.4 JADE: The Java Agent DEvelopment Framework

JADE can be considered an agent middle-ware that implements an Agent Platform and a development framework. It deals with all those aspects that are not peculiar of the agent internals and that are independent of the applications, such as message transport, encoding and parsing, or agent life-cycle.[10]

JADE offers a list of features to the agent programmer such as:[11]

- Distributed agent platform. The agent platform can be split on several hosts (provided that there is no firewall between them).
- Programming interface to simplify registration of agent services with one, or more, domains.
- Transport mechanism and interface to send/receive messages to/from other agents.
- Light-weight transport of messages inside the same agent platform.
- Graphical user interface to manage several agents and agent platforms from the same agent. The activity of each platform can be monitored and logged.

The software architecture is based on the coexistence of several Java Virtual Machines (VM) and communication relies on Java RMI (Remote Method Invocation) between different VMs and event signaling within a single VM. Each VM is a basic container of agents that provides a complete run time environment for agent execution and allows several agents to concurrently execute on the same host.

2.5 Remaining Research

The problem solving model outlined above can be applied to AIBO and might be usable for the collaboration between different AIBOs. However, using JADE on AIBO is not an option. Resources on AIBO are too limited for a full 'agent framework', specifically if written in a rather slow language such as JAVA. Besides, there is no Java Virtual Machine available for AIBO, making it impossible to run Java-based applications on it.

Since it turned out that JADE is not an option, an own transport mechanism for sending and receiving messages, and a full 'discussion protocol' needs to be developed in order to let the AIBOs communicate.

If the resources on the AIBO turn out to be sufficient, the possibility of sharing 'knowledge' between agents and maybe even 'keeping history' can be further investigated.

©2008 Delft University of Technology

©2008 Delft University of Technology

Communication Protocols 3

In order to let the AIBOs collaborate, some form of communication and therefore a protocol is needed. In the field of telecommunications, a communication protocol is the set of standard rules for data representation, signaling, authentication, and error detection required to send information over a communication channel.[12] The same definition will be used throughout this chapter.

Protocols may be implemented in hardware, software, or a combination of the two. At the lowest level, a protocol defines the behavior of a hardware connection. It is difficult to generalize about protocols because they vary so greatly in purpose and sophistication. Most protocols specify one or more of the following behaviors:

- detection of the underlying physical connection (wired or wireless), or the existence of the other endpoint or node
- handshaking
- negotiation of various connection characteristics
- how to start and end a message
- how to format a message
- what to do with corrupted or improperly formatted messages (error correction)
- how to detect unexpected loss of the connection, and what to do next
- termination of the session or connection.

3.1 Network Protocol Design

A protocol specification consists of five distinct parts. To be complete, each specification should include explicitly:

1. the service to be provided by the protocol
2. the assumptions about the environment in which the protocol is executed
3. the vocabulary of messages used to implement the protocol
4. the encoding (format) of each message in the vocabulary
5. the procedure rules guarding the consistency of message exchanges.

Below is an overview of a general set of principles of sound design. It is important to recognize that all these notes are variations on two common themes: simplicity and modularity.[13]

3.1.1 Simplicity

A well-structured protocol can be built from a small number of well-designed and well-understood pieces. Each piece performs one function and performs it well. To understand the working of the protocol it should suffice to understand the working of the pieces from which it is constructed and the way in which they interact. Protocols that are designed in this way are easier to understand and easier to implement efficiently, and they are more likely to be verifiable and maintainable. A light-weight protocol is simple, robust, and efficient.

3.1.2 Modularity

A protocol that performs a complex function can be built from smaller pieces that interact in a well-defined and simple way. Each smaller piece is a light-weight protocol that can be separately developed, verified, implemented, and maintained. Orthogonal functions are not mixed; they are designed as independent entities. The individual modules make no assumptions about each others working, or even presence. Error control and flow control, for instance, are orthogonal functions. They are best solved by separate light-weight modules that are completely unaware of each others existence. They make no assumptions about the data stream other than what is strictly necessary to perform their function. An error-correction scheme should make no assumptions about the operating system, physical addresses, data encoding methods, line speeds, or time of day. Those concerns, should they exist, are placed in other modules, specifically optimized for that purpose. The resulting protocol structure is open, extendible, and rearrangeable without affecting the proper working of the individual components.

3.1.3 Effectiveness

For a protocol to be effective it needs to be specified in such a way that engineers, designers, and software developers can implement and/or use it. In human-machine systems, its design needs to facilitate routine usage by humans. Protocol layering accomplishes these objectives by dividing the protocol design into a number of smaller parts, each of which performs closely related sub-tasks, and interacts with other layers of the protocol only in a small number of well-defined ways.

Protocol layering allows the parts of a protocol to be designed and tested without a combinatorial explosion of cases, keeping each design relatively simple. The implementation of a sub-task on one layer can make assumptions about the behavior and services offered by the layers beneath it. Thus, layering enables a 'mix-and-match' of protocols that permit familiar protocols to be adapted to unusual circumstances.

An example for this is TCP/IP. A sub-task of error detection and either message correction or retransmission may be performed by the Transmission Control Protocol

(TCP), while the related sub-task of addressing is implemented by the Internet Protocol(IP). TCP/IP may assume some point-to-point connectivity offered by point-to-point protocol (PPP) implemented in the lower-level Data link layer. At the lowest level is the sub-task involving the electrical encoding/decoding of bits into/from voltages performed in the Physical layer.

This example motivates the need to specify some software architecture or reference model that systematically places each subtask into its proper context. The reference model usually used for protocol layering is the OSI seven layer model, which can be applied to any protocol, not just the OSI protocols initially sanctioned by the ISO. This reference model also provides an opportunity to teach more general software engineering concepts like hiding, modularity, and delegation of tasks.

The seven layers of the OSI model and the general purpose of each are shown in figure 3.1.[14]

3.1.4 Reliability

Assuring reliability of data transmission involves error detection and correction, or some means of requesting retransmission.

Communication systems correct errors by selectively resending bad parts of a message. For example, in TCP (the Internet's Transmission Control Protocol), messages are divided into packets, each of which has a checksum. When a checksum is bad, the packet is discarded. When a packet is lost, the receiver acknowledges all of the packets up to, but not including the failed packet. Eventually, the sender sees that too much time has elapsed without an acknowledgment, so it resends all of the packets that have not been acknowledged. At the same time, the sender backs off its rate of sending, in case the packet loss was caused by saturation of the path between sender and receiver.

3.1.5 Resiliency

Resiliency addresses a form of network failure known as topological failure in which a communications link is cut, or degrades below usable quality. Most modern communication protocols periodically send messages to test a link. In phones, a framing bit is sent every 24 bits on T1 lines. In phone systems, when 'sync is lost', fail-safe mechanisms reroute the signals around the failing equipment.

In packet switched networks, the equivalent functions are performed using router update messages to detect loss of connectivity.

3.1.6 Robustness

It is not difficult to design protocols that work under normal circumstances. It is the unexpected that challenges them. It means that the protocol must be prepared to deal appropriately with every feasible action and with every possible sequence of actions under all possible conditions. The protocol should make only minimal assumptions about its environment to avoid dependencies on particular features that could change. Many link-level protocols that were designed in the 1970s, for instance, no longer work properly if

©2008 Delft University of Technology

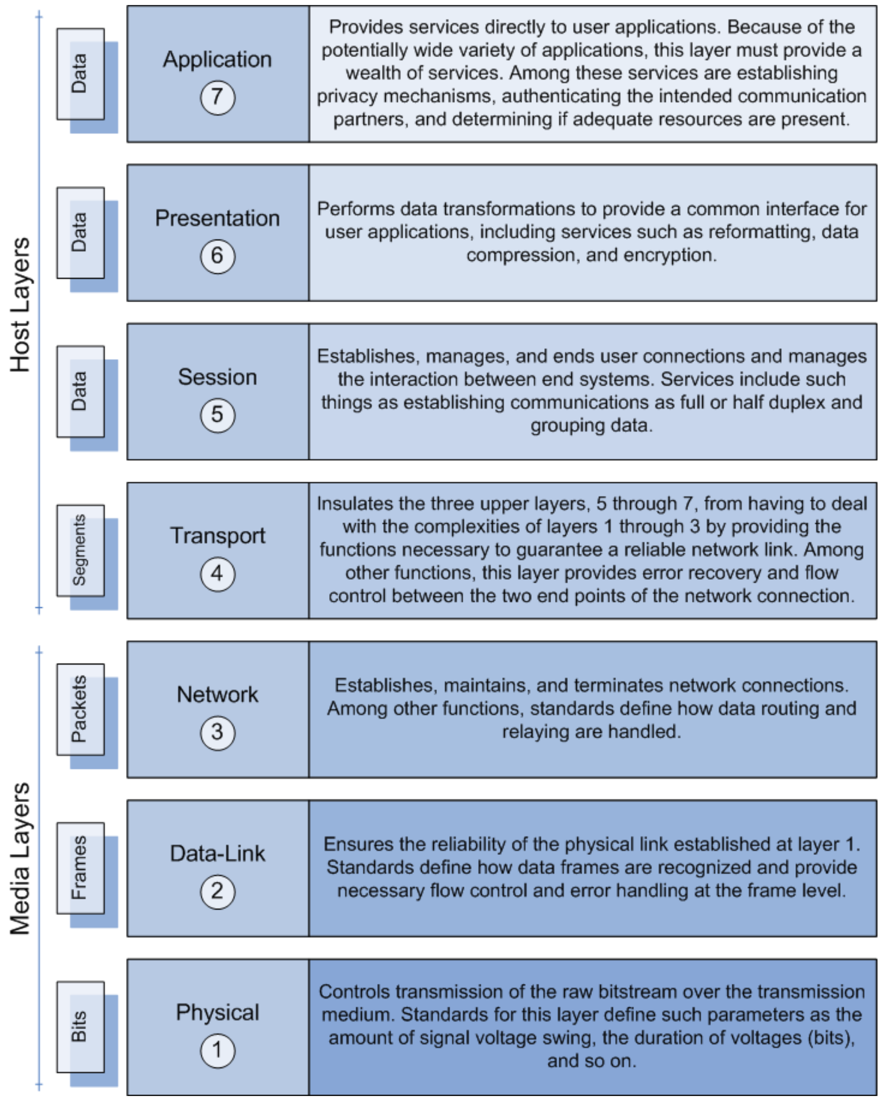

Host Layers
Media Layers
Data
Application 7
Provides services directly to user applications. Because of the potentially wide variety of applications, this layer must provide a wealth of services. Among these services are establishing privacy mechanisms, authenticating the intended communication partners, and determining if adequate resources are present.
Data
Presentation 6
Performs data transformations to provide a common interface for user applications, including services such as reformatting, data compression, and encryption.
Data
Session 5
Establishes, manages, and ends user connections and manages the interaction between end systems. Services include such things as establishing communications as full or half duplex and grouping data.
Segments
Transport 4
Insulates the three upper layers, 5 through 7, from having to deal with the complexities of layers 1 through 3 by providing the functions necessary to guarantee a reliable network link. Among other functions, this layer provides error recovery and flow control between the two end points of the network connection.
Packets
Network 3
Establishes, maintains, and terminates network connections. Among other functions, standards define how data routing and relaying are handled.
Frames
Data-Link 2
Ensures the reliability of the physical link established at layer 1. Standards define how data frames are recognized and provide necessary flow control and error handling at the frame level.
Bits
Physical 1
Controls transmission of the raw bitstream over the transmission medium. Standards for this layer define such parameters as the amount of signal voltage swing, the duration of voltages (bits), and so on.

Figure 3.1: The OSI model

they are used on very high speed data lines (in the Gigabits/sec range). A robust design automatically scales up with new technology, without requiring fundamental changes. The best form of robustness, then, is not over-design by adding functionality for anticipated new conditions, but minimal design by removing non-essential assumptions that could prevent adaptation to unanticipated conditions.

3.1.7 Consistency

There are some standard and dreaded ways in which protocols can fail. The three more important ones are ‘Deadlocks’, ‘Livelocks’, and ‘Improper terminations’:

- Deadlocks - states in which no further protocol execution is possible, for instance because all protocol processes are waiting for conditions that can never be fulfilled.
- Livelocks - execution sequences that can be repeated indefinitely often without ever making effective progress.
- Improper terminations - the completion of a protocol execution without satisfying the proper termination conditions.

In general, the observance of these criteria cannot be verified by a manual inspection of the protocol specification. More powerful tools are needed to prevent or detect them.

3.2 Well-formed Protocols

A well-formed protocol is not over-specified, that is, it does not contain any unreachable or unexecutable code. A well-formed protocol is also not under-specified or incomplete. An incompletely specified protocol, for instance, may cause unspecified receptions during its execution. An unspecified reception occurs if a message arrives when the receiver does not expect it, or cannot respond to it.

A well-formed protocol is bounded. It cannot overflow known system limits, such as the limited capacity of message queues.

A well-formed protocol is self-stabilizing. If a transient error arbitrarily changes the protocol state, a self-stabilizing protocol always returns to a desirable state within a finite number of transitions, and resumes normal operation. Similarly, if such a protocol is started in an arbitrary system state, it always reaches one of the intended states within finite time.

A well-formed protocol, finally, is self-adapting. It can adapt, for instance, the rate at which data are sent to the rate at which the data links can transfer them, and to the rate at which the receiver can consume them. A rate control method, for instance, can be used to change either the speed of a data transmission, or its volume.

3.3 Ten Rules of Design

The principles discussed above lead to ten basic rules of protocol design.[13]

1. Make sure that the problem is well-defined. All design criteria, requirements, and constraints should be enumerated before a design is started.
2. Define the service to be performed at every level of abstraction before deciding which structures should be used to realize these services (what comes before how).
3. Design external functionality before internal functionality. First consider the solution as a black-box and decide how it should interact with its environment. Then

decide how the black-box can internally be organized. Likely it consists of smaller black-boxes that can be refined in a similar fashion.

4. Keep it simple. Fancy protocols are buggier than simple ones; they are harder to implement, harder to verify, and often less efficient. There are few truly complex problems in protocol design. Problems that appear complex are often just simple problems huddled together. The job as designer is to identify the simpler problems, separate them, and then solve them individually.
5. Do not connect what is independent. Separate orthogonal concerns.
6. Do not introduce what is immaterial. Do not restrict what is irrelevant. A good design solves a class of problems rather than a single instance.
7. Before implementing a design, build a high-level prototype and verify that the design criteria are met.
8. Implement the design, measure its performance, and if necessary, optimize it.
9. Check that the final optimized implementation is equivalent to the high-level design that was verified.
10. Don't skip rules 1 to 7.

The most frequently violated rule, clearly, is rule 10.

3.4 Protocol Families

A number of protocol stacks or families exist. A few well known (both proprietary and open) protocols are summed below:

- *Proprietary standards:*
  - AppleTalk
  - DECnet
  - IPX/SPX
  - SMB
  - Systems Network Architecture (SNA)
  - Distributed Systems Architecture (DSA)
- *Open standards:*
  - Open Systems Interconnect (OSI)
  - Internet Protocol suite (TCP/IP).

These protocols work on different layers of the OSI model. The predominant standards at layer 7 and layer 6 were developed by the Department of Defense (DoD) as part of the Transmission Control Protocol/Internet Protocol (TCP/IP) suite. This suite consists of more than 40 protocols at several layers of the OSI model, such as:[14]

- File Transfer Protocol (FTP), the protocol most often used to download files from the Internet.
- Telnet, to connect to mainframe computers over the Internet.
- HyperText Transfer Protocol (HTTP), which delivers Web pages.
- Simple Mail Transfer Protocol (SMTP), which is used to send e-mail messages.

SMB is a proprietary protocol from Microsoft, and is used to share files and printers. It also works at layer 7 and 6, and can use a variety of underlying protocols for transportation. The most popular are 'NetBIOS over NetBEUI' and 'NetBIOS over TCP/IP'

IBM's NetBIOS (Network Basic Input/Output System) works at layer 4 and 5. Because it is designed specifically for a single network, this protocol does not support a routing mechanism to allow messages to travel from one network to another. For routing to take place, NetBIOS can be used in conjunction with another transport mechanism such as TCP. TCP provides all functions required for the transport layer.

Probably the most important layer 3 standard is Internet Protocol (IP), another part of the TCP/IP suite. This protocol is the basis for the Internet and for all intranet technology. IP has also become the standard for many LANs.

The most commonly used layer 2 and layer 1 protocols are those specified in the Institute of Electrical and Electronics Engineering (IEEE): 802.2 Logical Link Control, 802.3 Ethernet, 802.4 Token Bus, and 802.5 Token Ring. Most PC networking products use one of these standards.

Another layer 2 standard is Cells In Frames (CIF), which provides a way to send Asynchronous Transfer Mode (ATM) cells over legacy LAN frames.[14]

3.5 Protocol for AiboCommOpenR

The protocol that is needed for AiboCommOpenR can use UDP for the transmission of the data, because it is already available on AIBO, and supports broadcasting of data, which is an important aspect of AiboCommOpenR. Because UDP fulfills level 4 and 5 of the OSI model, the protocol should work on level 6 and 7, and needs to take care of the serializing and unserializing of the data that is transported between the AIBOs. A library must be written which takes care of that, so that other modules can link to it and use AiboCommOpenR. All other layers for the communication protocol of AiboCommOpenR can be fulfilled with existing protocols as described in chapter 3.4.

©2008 Delft University of Technology

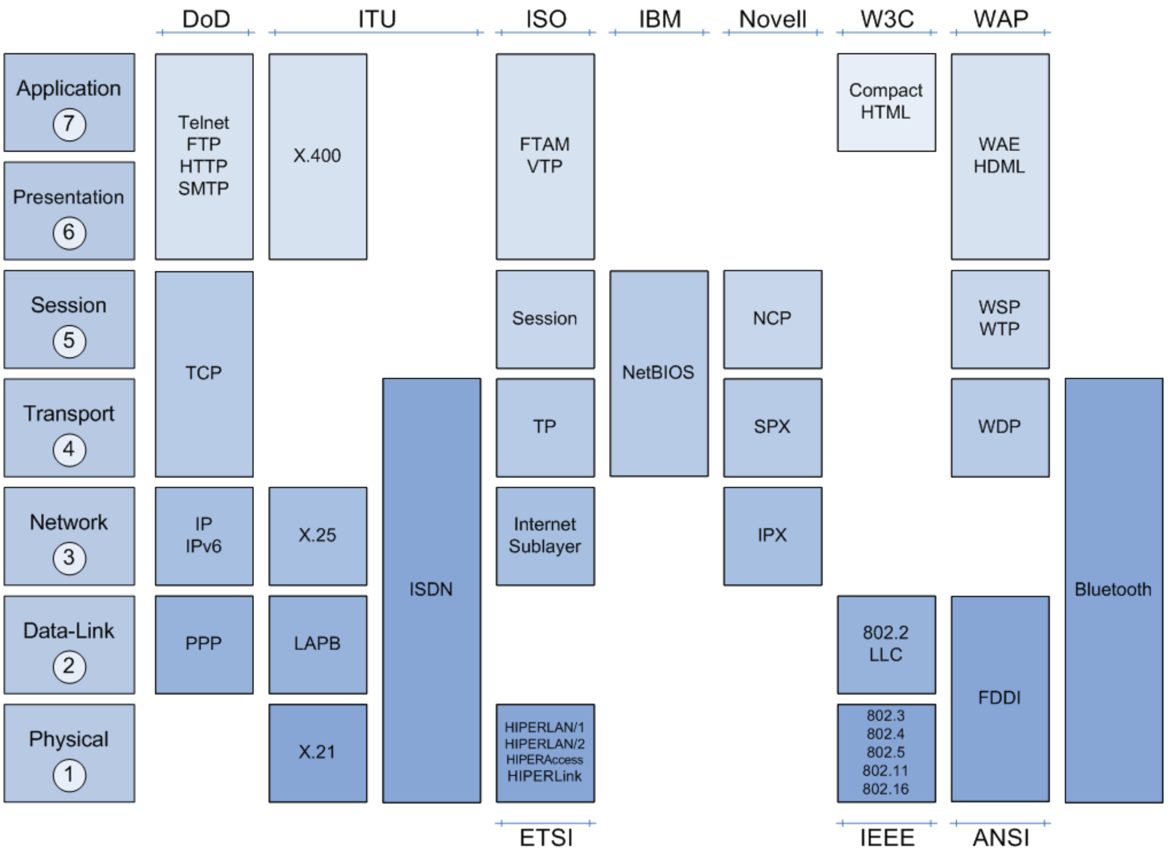


Figure 3.2: Important standards at various OSI layers

Because UDP is used as transport protocol, packets may get lost, reordered, and even duplicated. AiboCommOpenR should take care of these issues, so no deadlocks can occur while waiting for data that might be lost.

Choosing the Right Development Platform 4

In order to make AIBO an autonomous robot that can make decisions without the help of a stand-alone PC, a module has to be developed that runs on the AIBO itself. A variety of development platforms are available for developing such a module. This chapter outlines the limitations and benefits of these platforms for this project.

4.1 URBI: Universal Real-time Behavior Interface

> "URBI (Universal Real-time Behavior Interface) is a scripted language designed to work over a client/server architecture in order to remotely control a robot or, in a broader definition, any kind of device that has actuators and sensors."[15]

The main characteristics of URBI are:

- It is a low level command language. Motors and sensors are directly read and set. Although complex high level commands and functions can be written with URBI, the raw kernel of the system is low level by essence.
- It includes powerful time oriented control mechanisms to chain commands, serialize them, or build complex motor trajectories.
- It is designed to be independent from both the robot and the client system. It relies on TCP/IP, or 'Inter-Process Communication' if the client and the server are both running on board.
- It is designed with a constant care for simplicity. There is no 'philosophy' or 'complex architecture' to be familiar with. It is understandable in a few minutes and can be used immediately.

URBI consists of a server which is available for both AIBO and Linux, and of 4 different clients which can be programmed in 2 different languages:

- liburbi-cpp *(programming language: C++, Linux)*
- liburbi-cpp-windows *(programming language: C++, WIN32)*
- liburbi-java *(programming language: JAVA)*
- liburbi-OPENR *(programming language: C++).*

Since this project focuses on having an autonomous agent, both the server and the client have to run on the AIBO. Therefore, the only option is liburbi-OPENR, because there is no Java Virtual Machine available for Aperios, which is the operating system running on AIBO. Aperios can only use programs written in C/C++.

URBI has the advantage that the same written C++ code can compile as the client for a normal PC as well as the client for the robot simply by changing the urbi-library. Only a few things need to be taken into account. No threads, semaphores, or any other function call not available under OPEN-R can be used and the application has to end with `urbi::execute`, instead of having an infinite loop in the main function.

Using the library is fairly easy. Simple commands exist that move the joints of the robot and read the current values of different parameters.

Listing 4.1: Mirror the right front leg to the left front leg

```
legLF.load = 0; // make Left Front leg joints loose
while (true) {
   legRF1.val = legLF1.val &
   legRF2.val = legLF2.val &
   legRF3.val = legLF3.val
};
```

Since moving joints is not really part of this research, it is important to take a closer look at the networking capabilities of URBI.

URBI uses one main class called ‘UClient’ which contains 2 functions that are needed to send messages between different UClient instances (one instance running on each AIBO):

- *setCallback:* void setCallback(UCallbackListener listener, String tag)

  Registers a callback associated with the given tag.

  Parameters:

listener:	the registered callback.
tag:	the tag associated with the registered callback.

- *send:* int send(String arg)

  Sends a string to the other UClient.

  Parameters:

  arg: the string to send.

Listing 4.2 is a simple example using the UClient and the functions above.

Listing 4.2: From examples: 'URBIPing'[15]

```
#include <uclient.h>

...

UClient *c;

UCallbackAction pong(const UMessage & msg) {

   unsigned int ptime=msg.client.getCurrentTime() - sendtime;

   ...

   return URBI_CONTINUE;
}

int main(int argc, char * argv[]) {

   if (argc<2) {
      printf("usage: %s robot [msinterval] [count]\n",argv[0]);
          exit(1);
   }
   rname=argv[1];
   c=new UClient(argv[1]);
   c->start();

   ...

   c->setCallback(&pong,"uping");

   ...

   for (int i=0;i<count || (!count);i++) {

      ...

      sendtime = c->getCurrentTime();
      c->send("uping:ping;");
   }

   ...
}
```

The only drawback with this method is that the exact IP address of the other party (UClient) must be known, and that a broadcast can't be used. However using a fixed list with IP addresses is an option here because of the simplicity of liburbi. Communication in plain OPEN-R is more difficult (as described later in this chapter), so the overhead of using a fixed list with addresses is compensated here.

4.2 Tekkotsu

"Tekkotsu means 'iron bones' in Japanese, often used in the context of buildings' structural framework. Similarly, this software package aims to give you a structure on which to build."[16]

Tekkotsu builds on the basic functionality provided by the OPEN-R operating system. It is written in C++, (like the underlying system APIs) and makes full use of inheritance and templates. There is a delicate balance between ease of programming and speed of execution.

At its lowest level, Tekkotsu provides primitives for sensor processing, smooth control of effectors, and event-based communication. Higher level facilities include a hierarchical state machine formalism for managing control flow in the application, and an automatically maintained world map. Tekkotsu also provides housekeeping and utility functions useful to application developers, and a set of remote monitoring tools to permit real-time monitoring of various aspects of the robot's state.

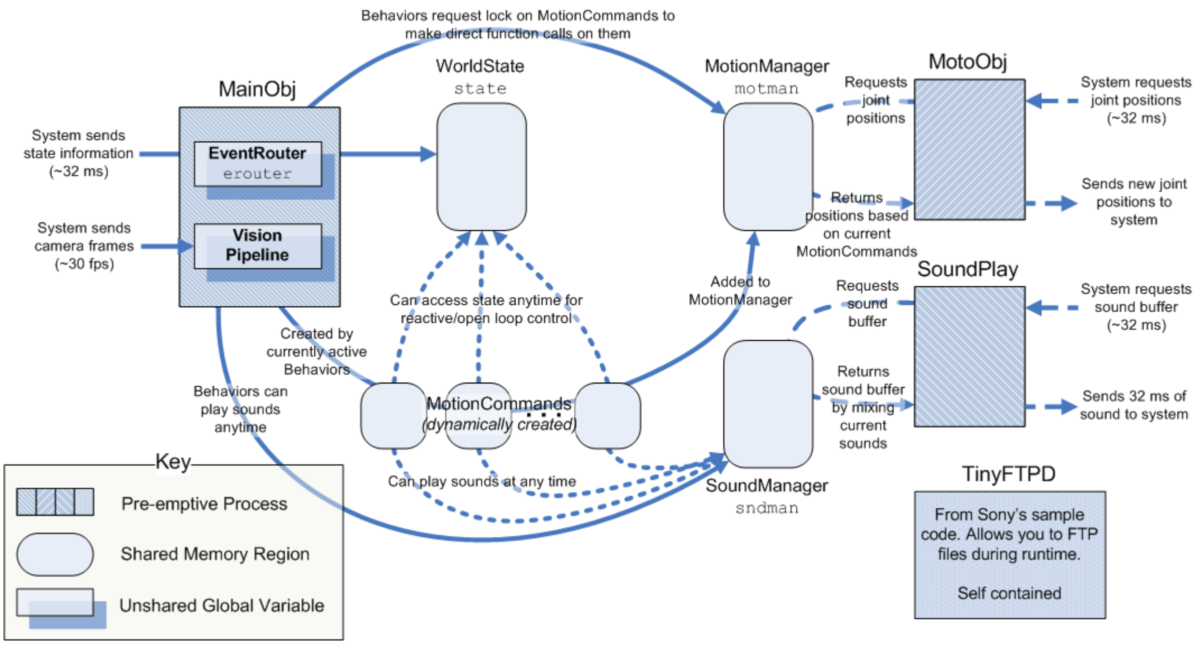

Behaviors request lock on MotionCommands to make direct function calls on them
MainObj
EventRouter
erouter
Vision Pipeline
System sends state information (~32 ms)
System sends camera frames (~30 fps)
WorldState
state
MotionManager
motman
Requests joint positions
Returns positions based on current MotionCommands
MotoObj
System requests joint positions (~32 ms)
Sends new joint positions to system
SoundPlay
Requests sound buffer
Returns sound buffer by mixing current sounds
System requests sound buffer (~32 ms)
Sends 32 ms of sound to system
Added to MotionManager
Can access state anytime for reactive/open loop control
Created by currently active Behaviors
Behaviors can play sounds anytime
MotionCommands (dynamically created)
Can play sounds at any time
SoundManager
sndman
TinyFTPD
From Sony's sample code. Allows you to FTP files during runtime.
Self contained
Key
Pre-emptive Process
Shared Memory Region
Unshared Global Variable

Figure 4.1: Tekkotsu object overview

A Tekkotsu application is organized as a collection of Behaviors and MotionCommands. Their member functions run in two cooperating processes, 'Main' and 'Motion'. Separate processes are used because they require different execution styles. Main handles perception and decision making. These are deliberative processes where each operation can potentially take a substantial amount of time. Motion is concerned with realtime control of effectors. All operations must be guaranteed to complete quickly, because several may have to be performed within a 32 msec timeslice. A third process, SoundPlay, handles audio output, and operates under the same realtime constraints as Motion. The processes are labeled MainObj, MotoObj, and SoundPlay in figure 4.1.

Behaviors are instantiated and run exclusively in the Main process' address space (green hatching in figure 4.1). MotionCommands are instantiated in shared memory (solid blue in the figure). Most of their computation is done in the Motion process (orange hatching), but certain member functions run in Main. A clever mutual exclusion mechanism called 'MMAccessor' provides Behaviors running in Main with a safe way to

©2008 Delft University of Technology

update shared memory structures while they are in use by the MotionManager running in Motion.

Where a lot of work is done in the motion part of AIBO, the communication part (which is of interest in this thesis project) is fairly limited. A socket class is available but is not really usable for communication between AIBOs, because it is to complex and was written for communication between a GUI and the AIBO. The developers of Tekkotsu also state that the networking code needs a complete rewrite:

> "The networking code is actually scheduled for an overhaul. It definitely has some limitations, although it probably is easier to use than the OPEN-R code it is hiding. Limitations to be aware of – the buffer size you set when initializing a socket is the maximum you can send at a time. Probably not a big issue for robosoccer. You don't want to be sending huge amounts of data between AIBOs anyway. More annoying though is currently no support for 'server sockets', meaning incoming connections are bound on the port they connected to, thus preventing other AIBOs from connecting on the same port. So your AIBOs each need to connect to a different port on each other. Not a huge limitation, as long as you plan for it."
>
> – Ethan Tira-Thompson <*ejt@andrew.cmu.edu*>, in a personal e-mail

Especially the last part about 'server sockets' is a problem for this project. Keeping a list of ports and distributing them between the AIBOs seems like a waste of the already scarce resources. It is much easier if all the AIBOs have only one port on which they listen for incoming messages from all other AIBOs.

4.3 AIBO SDE: AIBO Software Development Environment

> "The AIBO SDE is a full-featured development environment where you can make software for AIBO. The AIBO SDE contains the OPEN-R SDK, the R-CODE SDK, the AIBO Remote Framework and the AIBO Motion Editor. [...] OPEN-R is the standard interface for the entertainment robot system that SONY is actively promoting. This interface greatly expands the capabilities of entertainment robots." [17]

The AIBO SDE is the 'AIBO Software Development Environment' that can make software that either executes on AIBO, or executes on a PC and controls AIBO by using a wireless LAN. AIBO SDE contains three SDKs (Software Development Kits), and one motion editor (AIBO Motion Editor). The three SDKs are named OPEN-R SDK, R-CODE SDK, and AIBO Remote Framework.

- The OPEN-R SDK is a cross-development environment based on gcc (C++), with which software can be written that runs on AIBO.

- The R-CODE SDK is an environment which can execute programs written in R-CODE, a scripting language, on AIBO.
- The AIBO Remote Framework is a Windows PC application development environment based on Visual C++, with which software can be written that runs on a Windows PC. The software can control AIBO remotely via wireless LAN.
- AIBO Motion Editor is a motion creation editor for AIBO. Motions created with AIBO Motion Editor can be used with the OPEN-R SDK, R-CODE SDK, and AIBO Remote Framework.

Characteristics of the AIBO SDE:
Three Software Development Kits for different uses.

- The OPEN-R SDK is typically used for research in robotics programming.
- The R-CODE SDK is used with an easy-to-understand scripting language. A complete R-CODE program can consist of only a few lines of code.
- AIBO Remote Framework is used to create rich PC programs that interact with AIBO. For example, streaming camera data and microphone audio can be sent to a nearby PC, and streaming audio data can be send back to AIBO as well. Taking advantage of the PC's power and connectivity can lead to feature-rich commercial applications.

Because of the wireless LAN capabilities of AIBO, and the development a stand-alone program which runs on the AIBO itself are important for this thesis project, the 'OPEN-R SDK' has to be used.

The OPEN-R SDK discloses the specifications of the interface between the 'system layer' and the 'application layer'. OPEN-R software is object-oriented and modular. Software modules are called 'objects' (specifically, 'OPEN-R objects'). In OPEN-R, robot software is implemented so that processing is performed by multiple objects with various functionality running concurrently, communicating with each other via inter-object communication. Concurrency is achieved by giving running objects time slices in which to perform a task before being preempted or forced to yield to the next object that is ready to run.

The OPEN-R system layer provides a set of services (input of sound data, output of sound data, input of image data, output of control data to joints, and input of data from various sensors) as an interface to the application layer. This interface is also implemented by inter-object communication. These services enable application objects to utilize the robots underlying functionality, without requiring detailed knowledge of the hardware devices that comprise the robot. The system layer also provides an interface for the TCP/IP protocol stack, which enables programmers to create networking applications utilizing the wireless LAN. For this, a complete IPv4 protocol stack is available.[18]

OPEN-R SDK has an extra feature called 'Remote Processing OPEN-R' (RP_OPEN-R), which is a remote processing environment where an OPEN-R based program is executed on a machine other then AIBO. By using RP_OPEN-R, some objects can be executed on a remote host (connected to AIBO via wireless LAN), and the other objects can be executed on AIBO directly. There is no concern about which object is running on which host, since all the communication is done transparently using inter-object communication. 'Remote Processing OPEN-R' has the advantage that the high performance and versatile functionality of the remote host can be used, and makes it easier and more efficient to develop programs and perform debugging. However, since this thesis project focuses on a stand-alone module, it is of little use here.

Because the IP stack communicates through message passing with objects and with device drivers, and because of the complexity of this inter-object communication, no sample-code of OPEN-R wireless LAN communication will be given here. Object communication will be further discussed in chapter 5.1.

4.4 Final Assessments

Having all the pros and cons in mind, the right development platform for this project can be chosen. While URBI seems to be the best candidate, it can't be used here, simply because at the time of starting with this project, 'liburbi-OPENR' is not available yet.

Tekkotsu's networking capabilities are too limited, and having all the modules around for motion while not using them seems to be too much overhead.

Therefore, OPEN-R remains as the final candidate for this project and will be the platform of choice. It has the most advanced networking capabilities, and is the fastest in runtime. Because of the complexity of object communication, the next chapter (5) will describe it in detail.

Table 4.1: Development platform overview

Platform	**Language**	**Ease of Use**	**Networking Code**	**Comments**
URBI	Java/C++	Medium	Simple to use	- *Pro:* Ease of programming - *Pro:* Simple networking code - *Pro:* Different programming languages - *Con:* liburbi-OPENR is not available yet
Tekkotsu	C++	Medium	Not usable	- *Pro:* Ease of programming - *Con:* Limited networking capabilities - *Con:* Lot of overhead (sound/vision)
R-CODE SDK	Scripting	Easy	Not available	- *Pro:* Very easy to use - *Con:* Lack of networking capabilities
OPEN-R SDK	C++	Difficult	Difficult to use	- *Pro:* Fastest in runtime - *Pro:* Advanced networking capabilities - *Con:* Difficult to use

©2008 Delft University of Technology

5 OPEN-R Objects

This chapter will discuss 'OPEN-R Objects' and especially the communication between them. If the word 'object' is mentioned in this chapter, it means an 'OPEN-R object' and not an object in the traditional C++ (Object Oriented Programming) sense of the word.

In OPEN-R, multiple objects with various functionality are running concurrently and communicate with each other via inter-object communication (see figure 5.1). The concept of an object in OPEN-R is similar to one of a process in the UNIX or Windows operating systems.

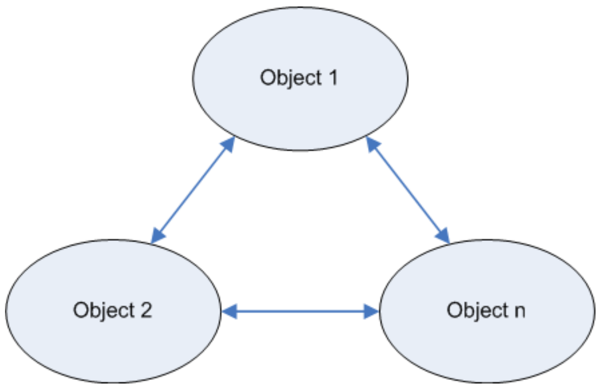


Figure 5.1: OPEN-R Application software

An object is a concept that only exists at run-time. Each object has a counterpart in the form of an executable file, created at compile-time. Source code is compiled and linked (with the use of the OPEN-R SDK) to create this executable file. The file is put on an AIBO Programming Memory Stick and when AIBO boots, the operating system loads the file from the AIBO Programming Memory Stick and executes it as an object.[18]

Unlike an ordinary programming environment in which a program has a single entry point '`main()`', OPEN-R allows an object to have multiple entry points. Some entry points have purposes that are determined by the system, e.g. initialization and termination. Other entry points have purposes specific to the object.

An object can send messages to other objects. A message contains some data and the selector, which is an integer that specifies the task to be done by the receiver of the message. When an object receives a message, the function corresponding to the selector is invoked, with the data in the message as its argument. A function corresponding to a selector is called a 'method'. An important feature of objects is that they are single-threaded. As a consequence, an object can process only one message at a time. If an object receives a message while it is processing another message, the second message is put into the message queue and processed later.

Below is the typical life cycle of an object:

1. loaded by the system
2. wait for a message
3. when a message arrives, execute the method corresponding to the selector specified in the message and send some messages to other objects when necessary
4. when the method finishes execution, go to step 2.

This is an infinite loop. An object cannot terminate itself and persists while the system is activated.

5.1 Inter-Object Communication

OPEN-R objects communicate with each other while they perform their tasks. In OPEN-R, this communication between objects is called 'inter-object communication'. The use of inter-object communication enables each object to be created separately and later be connected to other objects. When two objects communicate, the side that sends data is called the 'subject', and the side that receives data is called the 'observer'. Figure 5.2 shows a case where the subject of object A communicates with the observer of object B.

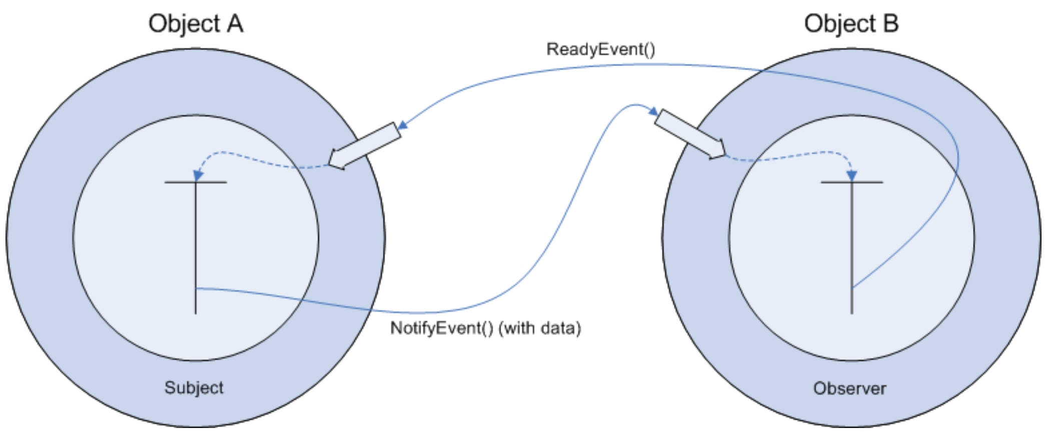


Figure 5.2: Inter-object communication

When the observer is in a state ready to receive data, the observer sends `ASSERT-READY` to the subject. When the observer is in a state not ready to receive data, the observer sends `DEASSERT-READY` to the subject. The subject receives these in

©2008 Delft University of Technology

the form of a `ReadyEvent`. When the function `IsAssert()` in the `ReadyEvent` returns `true`, the subject can send a `NotifyEvent` to the observer. `NotifyEvent` includes the data that the subject wants to send to the observer.

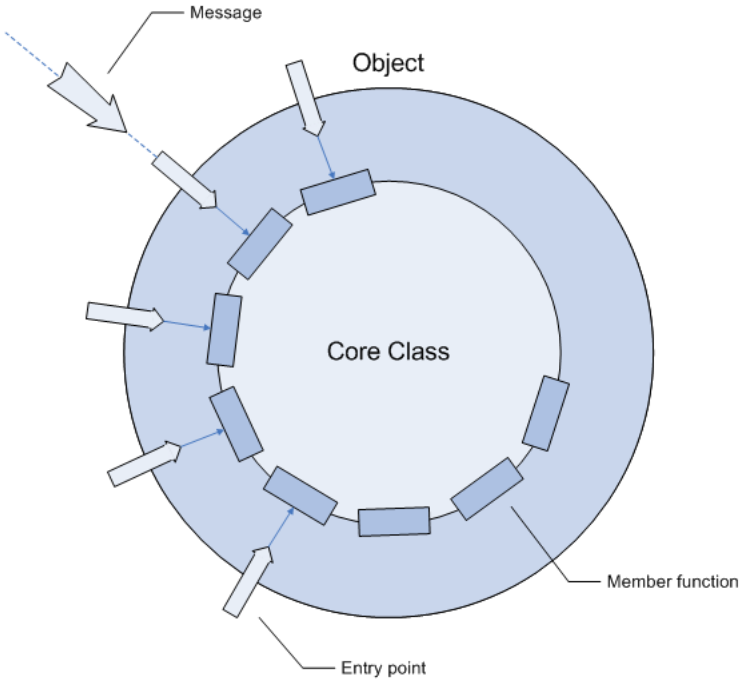


Figure 5.3: A core class

Objects have entry points for receiving messages. As shown in figure 5.3, each entry point corresponds to a particular method of the object, and each method corresponds to a particular member function of the object. In C++, an OPEN-R object can be represented by a core class. Core classes have the following characteristics:[18]

- A core class inherits from the OObject class (which is the base class for all objects).
- A core class implements the `DoInit()`, `DoStart()`, `DoStop()` and `DoDestroy()` functions.
- A core class has the necessary number of OSubject and OObserver (described in chapter 5.2).
- Some member functions in the core class correspond to specific methods in the object:
    1. Methods that are called at startup and shutdown:
        - *Init method*
          This is called at startup and initializes instances and variables.

   - *Start method*
     This is called at startup after Init is executed in all objects.
   - *Stop method*
     This is called at shutdown.
   - *Destroy method*
     This is called at shutdown after Stop is executed in all objects and destroys the subject and observer instances.

   The Init method, Start method, Stop method, and Destroy method correspond to each `DoInit()`, `DoStart()`, `DoStop()` and `DoDestroy()` function in the object's corresponding core class, respectively.

2. Methods that are called when a message is received from another object:
   Methods used in subjects:
   - *Control method*
     This receives the connection results between the subject and its observers.
   - *Ready method*
     The subject receives `ASSERT-READY` or `DEASSERT-READY` notifications from the observers.

   Methods used in observers:
   - *Connect method*
     This receives the connection results between an observer and its subjects.
   - *Notify method*
     This receives a message from the subject.

5.2 Configuration Files

A `stub.cfg` file is used to describe the entry points and member functions of each object. A tool from OPEN-R SDK (stubgen2) reads the `stub.cfg` file and generates stub files for each object. These stub files are needed for compilation purposes. The following items are described in `stub.cfg`:

- the number of subjects and the number of observers
- services used in inter-object communication.

The subjects and observers provide the services for inter-object communication. Each service has a unique name, in order to distinguish that service from other services in the system.

Below, the configuration files are given where one object is a 'observer' and one is a 'subject'.

Listing 5.1: Sample observer stub.cfg

```
ObjectName : SampleObserver
```

©2008 Delft University of Technology

```
NumOfOSubject   : 1
NumOfOObserver  : 1
Service : "SampleObserver.DummySubject.DoNotConnect.S", null, null
Service : "SampleObserver.ReceiveString.char.O", null,
    ReceiveString()
```

Listing 5.2: Sample subject stub.cfg

```
ObjectName : SampleSubject
NumOfOSubject   : 1
NumOfOObserver  : 1
Service : "SampleSubject.SendString.char.S", null, SendString()
Service : "SampleSubject.DummyObserver.DoNotConnect.O", null, null
```

Following are the descriptions of each item:

- **ObjectName** - the core class name.
- **NumOfOSubject** - this is the number of subjects. At least 1 subject must be specified. If none is available, a dummy subject should be registered (as shown on line 4 of listing 5.2).
- **NumOfOObserver** - this is the number of observers. At least 1 observer must be specified. If none is available, a dummy observer should be registered.
- **Service** - this is the communication service that the object provides. A service corresponding to each subject and observer is described. A service consists of the following items: *'(Connection name)', (Member function 1), (Member function 2).*
  - Connection name
    The connection name consists of the following items:
    (Object name).(Subname).(Data name).(Service type)
    * *Object Name*
      This can be any name, but is usually the core class name.
    * *Subname*
      This is the service name, which must be unique. Do not use the same subname for other services.
    * *Data name*
      This is the name corresponding to the data type used in inter-object communication. It accepts primitives as well as complex structures.
    * *Service type*
      This is the type of the service. S for subject or O for observer.
  - Member function 1
    This member function is called when a connection result is received. It should correspond to a function implemented in the core class. In case it is not needed, 'null' can be specified here.

©2008 Delft University of Technology

– Member function 2
If this service is for observers, this function is called when a message is received from a subject. If this service is for subjects, this function is called when `ASSERT-READY` or `DEASSERT-READY` is received from an observer. If this member function is left out (defined as 'null') as described in chapter 6.1.2, one can call the subject function by hand, and no function is called when the observer is ready.

The subject's service can be connected to the observer's service by describing both service names in `connect.cfg`. When the system software boots, this description file is loaded and used to allocate and configure the communication paths for inter-object communication. For the examples above, the two services are tied together by the following `connect.cfg`:

Listing 5.3: Sample connect.cfg

```
SampleSubject.SendString.char.S SampleObserver.ReceiveString.char.O
```

5.3 Example Code

5.3.1 When the Observer Activates the Subject

In listing 5.1, a `'member function 2'` is specified as `ReceiveString()`. This means that when the observer in the `SampleObserver` object is in the ready-state, the subject function in `SampleSubject` is automatically activated. This happens as soon as macro `ASSERT_READY_TO_ALL_OBSERVER` is called in `DoStart()` (see listing 5.4, line 20).

Listing 5.4: SampleObserver source

```
1  #include <OPENR/OSyslog.h>
2  #include <OPENR/core_macro.h>
3  #include "SampleObserver.h"
4
5  ...
6
7  OStatus
8  SampleObserver::DoInit(const OSystemEvent& event)
9  {
10     NEW_ALL_SUBJECT_AND_OBSERVER;
11     REGISTER_ALL_ENTRY;
12     SET_ALL_READY_AND_NOTIFY_ENTRY;
13     return oSUCCESS;
14 }
15
16 OStatus
17 SampleObserver::DoStart(const OSystemEvent& event)
18 {
19     ENABLE_ALL_SUBJECT;
20     ASSERT_READY_TO_ALL_OBSERVER;
21     return oSUCCESS;
22 }
```

```

...

void
SampleObserver::ReceiveString(const ONotifyEvent& event)
{
    const char* text = (const char *)event.Data(0);
    OSYSPRINT(("SampleObserver::Notify() %s\n", text));
    observer[event.ObsIndex()]->AssertReady();
}
```

Listing 5.5: SampleSubject source (with '`member function 2`' specified)

```
#include <string.h>
#include <OPENR/OSyslog.h>
#include <OPENR/core_macro.h>
#include "SampleSubject.h"

...

OStatus
SampleSubject::DoInit(const OSystemEvent& event)
{
    NEW_ALL_SUBJECT_AND_OBSERVER;
    REGISTER_ALL_ENTRY;
    SET_ALL_READY_AND_NOTIFY_ENTRY;
    return oSUCCESS;
}

OStatus
SampleSubject::DoStart(const OSystemEvent& event)
{
    ENABLE_ALL_SUBJECT;
    ASSERT_READY_TO_ALL_OBSERVER;
    return oSUCCESS;
}

...

void
SampleSubject::SendString(const OReadyEvent& event)
{
    OSYSPRINT(("SampleSubject::Ready() : %s\n",
               event.IsAssert() ? "ASSERT READY" : "DEASSERT READY
                   "));

    static int counter = 0;
    char str[32];

    if (counter == 0) {

        strcpy(str, "!!! Hello world !!!");
        subject[sbjSendString]->SetData(str, sizeof(str));
        subject[sbjSendString]->NotifyObservers();

    } else if (counter == 1) {

        strcpy(str, "!!! Hello world again !!!");
        subject[sbjSendString]->SetData(str, sizeof(str));
        subject[sbjSendString]->NotifyObservers();

    }

    counter++;
}
```

The corresponding stubs are in listing 5.1 and 5.2.

The flow of this application is as follows. Both objects are loaded into memory and `DoInit()` of both objects is called. After both `DoStart()` functions are called and `SampleObserver` 'asserts' his observers. This signal is captured by `SampleSubject` and entry-point `'void SampleSubject::SendString(const OReadyEvent& event)'` is activated. This function returns his data to `SampleObserver` using `SetData()` and `NotifyObservers()` (line 39, 40 of listing 5.5). `SampleObserver` receives this data with the use of `event.Data()` (line 29 of listing 5.4). After that he 're-asserts' himself using `AssertReady()` (line 31 of listing 5.4) and `SampleSubject` then sends the second string.

5.3.2 When the Subject is Called Manually

In the example given above, it is clear that the observer controls when data is send. In case it is necessary to have the subject initialize the process, we can specify `'member function 2'` as 'null' in the `stub.cfg` for the subject object (see listing 5.6, line 4) and use the following technique.

Listing 5.6: Sample subject stub.cfg (with no `'member function 2'`)

```
1 ObjectName : SampleSubject
2 NumOfOSubject   : 1
3 NumOfOObserver  : 1
4 Service : "SampleSubject.SendString.char.S", null, null
5 Service : "SampleSubject.DummyObserver.DoNotConnect.O", null, null
```

The `connect.cfg` is the same for both cases, and so is the observer stub. Only the source code of SampleSubject is different (see listing 5.7).

Listing 5.7: SampleSubject source (with no `'member function 2'`)

```
#include <string.h>
#include <OPENR/OSyslog.h>
#include <OPENR/core_macro.h>
#include "SampleSubject.h"

...

void
SampleSubject::SendString()
{

    static int counter = 0;
    char str[32];

    if (counter == 0) {

        strcpy(str, "!!! Hello world !!!");
        subject[sbjSendString]->SetData(str, sizeof(str));
        subject[sbjSendString]->NotifyObservers();

    }

    ...

}
```

Every function in `SampleSubject` can now call `SendString()` and send data to `SampleObserver`. Notice, however, that `SampleObserver` needs to have his observers ready! This can be done like in the example in listing 5.4 or by asserting a specific observer (notice the difference in line 20 of 5.4 and line 12 of 5.8).

Listing 5.8: SampleObserver source (with different ‘asserting’)

```
1  #include <OPENR/OSyslog.h>
2  #include <OPENR/core_macro.h>
3  #include "SampleObserver.h"
4
5  ...
6
7
8  OStatus
9  SampleObserver::DoStart(const OSystemEvent& event)
10 {
11     ENABLE_ALL_SUBJECT;
12     observer[obsReceiveString]->AssertReady();
13     return oSUCCESS;
14 }
15
16 ...
```

Using this technique, we can control the entire message-passing between the running objects. When the observer sends `ASSERT-READY`, the corresponding subject method won’t be activated. The observer will keep waiting until the subject sends the data. That moment can be controlled by calling the subject-function (`SendString()`) at any given time. Waiting for data is non-blocking, which means that the observer can send and receive data to and from other observers and subjects while waiting for data from a specific subject. After receiving the data from a subject, the observer needs to ‘re-assert’ himself again, and is then ready to receive new data.

When an observer is not ready to receive data, and a subject sends him a `NotifyEvent()`, this message will be ignored.

©2008 Delft University of Technology

6 AiboCommOpenR Module Overview

A number of issues have to be addressed in order to develop a successful distributed system for AIBO. These issues pertain to several different subject areas, such as a functioning motion system, which deals with mechanical robotics, as well as a functioning vision system, which deals with digital image analysis and color calibration. Furthermore, the system deals with artificial intelligence, distributed systems, real time systems and communication.

The focus for this project has been on the distribution and communication issue. The main goal is the design and implementation of a system that runs on the AIBO and operates concurrently with the modules that are already developed (e.g. the motion modules for the soccer game). The system consists of several modules which can be used to make distributed decisions with other AIBOs. It takes care of the reasoning and of the transmission of data between the AIBOs. The only thing that other modules have to do is:

1. Suggest an action to the AiboReasoning module.
2. Wait for a 'go' or 'no go' and act accordingly.

6.1 Design

As described above, the system has been divided into a number of modules according to functionality. The modules that make up AiboCommOpenR are:

- **AiboReasoning** - this is the module that does the actual reasoning, with the help of an expert system.
- **AiboCommIN** - listens for incoming data from other AIBOs.
- **AiboCommOUT** - this module is used by AiboReasoning to broadcast data to other AIBOs.
- **AiboDummyActionGenerator** - used as a test module in replacement of real modules that are used on AIBO (e.g. from the soccer game). The current implementation starts the whole sequence of events when the back-sensor of AIBO is touched, and blinks its head-LEDs when a 'go' is received.
- **PulseGenerator** - is used as a timer for AiboCommOUT when it is broadcasting data and waiting for feedback.

The communication between these modules is done using the 'observer/subject' structure described in chapter 5.1. An overview of the modules and how they communicate is presented in figure 6.1. An example of using the design is given in chapter 6.1.1.

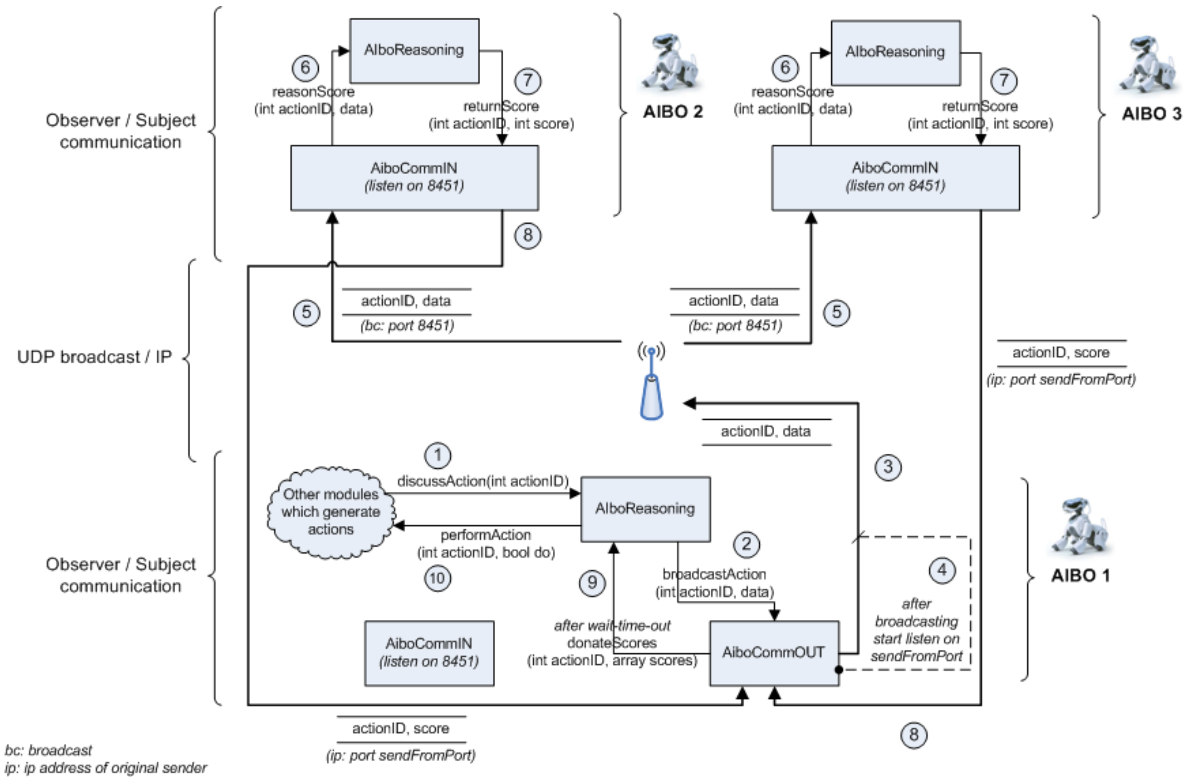

Observer / Subject communication
UDP broadcast / IP
Observer / Subject communication
AIBO 2
AIBO 3
AIBO 1
AIboReasoning
AiboCommIN (listen on 8451)
AiboCommOUT
6 reasonScore (int actionID, data)
7 returnScore (int actionID, int score)
8
5 actionID, data (bc: port 8451)
actionID, score (ip: port sendFromPort)
actionID, data
3
1 discussAction(int actionID)
Other modules which generate actions
performAction (int actionID, bool do)
10
2 broadcastAction (int actionID, data)
9 after wait-time-out donateScores (int actionID, array scores)
4 after broadcasting start listen on sendFromPort
bc: broadcast
ip: ip address of original sender

Figure 6.1: Module overview of AiboCommOpenR, showing the flow when AIBO 1 wants to perform an action

6.1.1 Example of Using AiboCommOpenR

This example gives the sequences of events that occur when AiboCommOpenR (as outlined in figure 6.1) is used in the soccer game example.

In figure 6.1, AIBO 1 is the initiator. The vision- and pattern recognition module on that AIBO finds the ball, and decides to move towards it. This might seem like a good idea from one AIBO's point of view, but if other AIBOs are closer to the ball, they should go to the ball and attack instead. This AIBO should then stay in the background and play defense. Using AiboCommOpenR for this example require the following steps:

1. The vision- and pattern recognition module detects the ball and decides to move towards it. However, before he acts, he gives a signal to `AiboReasoning` using `discussAction()`. He sends along the actionID for moving (which he gets from a list, distributed between all AIBOs) and the data, in this case the distance to the ball and his own position in the field. For this example we take 150cm for the distance.

   Because of the limited number of actions that AIBO can perform, we can use a list of actionIDs. Every AIBO in the field has a copy of this list.

2-4. `AiboReasoning` broadcasts the data and actionID to the other available AIBOs using the `AiboCommOUT` module and a call to `broadcastAction()`.

While the previous communication between the modules was done using the 'observer' and 'subject' technique, `AiboCommOUT` broadcasts the data and actionID using UDP. After broadcasting the information, `AiboCommOUT` starts listening for replies from the other AIBOs.

5-8. The other AIBOs (AIBO 2 and AIBO 3 in this case) have a `AiboCommIN` module listening on port 8451 and receive the information. They pass it to their own `AiboReasoning` module with a `reasonScore()` call. The received data, in combination with their own data and knowledge, is used by the reasoning engine for calculating a score for the idea of 'moving to the ball' of AIBO 1.

If AIBO 2 calculates a distance of himself to the ball of 200cm, he sends back a high score (e.g. *(8)*) to AIBO 1, since he is much further away from the ball. AIBO 3, in this example, finds a distance of 30cm to the ball, and therefore sends back a very low score (e.g. *(2)*). He does this by calling `returnScore()` in `AiboCommIN`. AIBO 1 receives these on his listening port in his `AiboCommOUT` module.

9. After receiving all the scores of the other AIBOs, or after waiting for a maximum amount of time (we don't want to stall the AIBO for to long), `AiboCommOUT` of AIBO 1 stops listening and returns the received scores to the `AiboReasoning` module using `donateScores()`. `AiboReasoning` receives the array of scores and based on these scores, it makes a final decision whether to give a 'go' or 'no go' to the vision- and pattern recognition module (which was the module that made the initial call to `discussAction()`. In this example it decides to give a 'no go', because the average of the received scores (*(8+2)/2 = 5*) results in a low score.

10. The vision- and pattern recognition module receives this advice (with the help of `performAction()`) and abandons the idea of moving towards the ball. After that it resumes his normal work.

Corresponding sequence diagrams, statechart diagrams, and a class diagram for AiboCommOpenR can be found in appendix A.

6.1.2 Using the 'Observer/Subject' Structure

When using the 'observer/subject' structure as in the design above, the observers and subjects need to be chosen carefully.

An observer sends a signal to the subject that it can receive information (as described in chapter 5.1):

> "When the subject receives `ASSERT-READY` from the observer, the subject starts to send data to the observer. The observer receives this data and when it is in a state ready to receive the next data, it sends `ASSERT-READY` again."[18]

Notice that the observer requests data from the subject. The opposite is required in the `AiboCommOUT` module. When `AiboReasoning` has all the data available that it wants to broadcast, it needs to send that to `AiboCommOUT`, calling the subject in `AiboReasoning` itself. This is possible by not defining a subject-function in the corresponding `stub.cfg` (see chapter 5.3.2), and call the subject like a normal function (which then sends the data). This is only possible if the corresponding observer in `AiboCommOUT` is ready to receive data! Therefore some sort of timing scheme needs to be designed to make sure that all the observers are ready when they should. Figure 6.2 and 6.3 show the scheme that is designed for AiboCommOpenR.

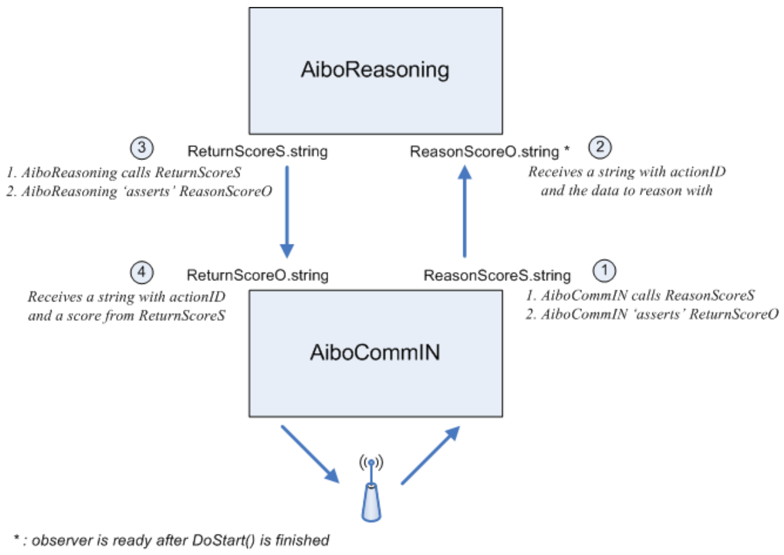

AiboReasoning
3
ReturnScoreS.string
1. AiboReasoning calls ReturnScoreS
2. AiboReasoning 'asserts' ReasonScoreO
ReasonScoreO.string *
2
Receives a string with actionID and the data to reason with
4
ReturnScoreO.string
Receives a string with actionID and a score from ReturnScoreS
ReasonScoreS.string
1
1. AiboCommIN calls ReasonScoreS
2. AiboCommIN 'asserts' ReturnScoreO
AiboCommIN
* : observer is ready after DoStart() is finished

Figure 6.2: 'Observer/Subject' structure of 'AiboCommOpenR, AiboCommIN'

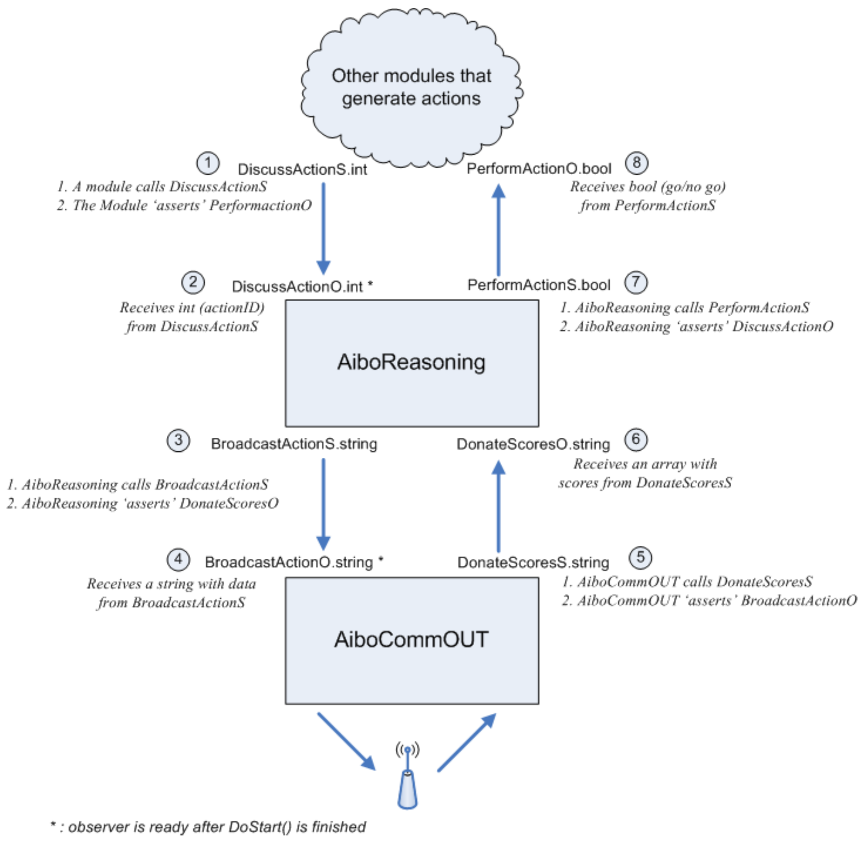

Other modules that generate actions
1 DiscussActionS.int
1. A module calls DiscussActionS
2. The Module 'asserts' PerformactionO
PerformActionO.bool 8
Receives bool (go/no go) from PerformActionS
2 DiscussActionO.int *
Receives int (actionID) from DiscussActionS
PerformActionS.bool 7
1. AiboReasoning calls PerformActionS
2. AiboReasoning 'asserts' DiscussActionO
AiboReasoning
3 BroadcastActionS.string
1. AiboReasoning calls BroadcastActionS
2. AiboReasoning 'asserts' DonateScoresO
DonateScoresO.string 6
Receives an array with scores from DonateScoresS
4 BroadcastActionO.string *
Receives a string with data from BroadcastActionS
DonateScoresS.string 5
1. AiboCommOUT calls DonateScoresS
2. AiboCommOUT 'asserts' BroadcastActionO
AiboCommOUT
* : observer is ready after DoStart() is finished

Figure 6.3: ‘Observer/Subject’ structure of ‘AiboCommOpenR, AiboCommOUT’

6.1.3 Communication Protocol

The communication in AiboCommOpenR is done in both the `AiboCommIN` and `AiboCommOUT` module. Sending data between these two modules involves serializing and unserializing the data using a predefined format. The following scheme is used for AiboCommOpenR:

```
actionID::data-element1::data-element2::data-element3
```

In `AiboCommOUT` the data elements are made up by data received from other modules. In case of the football example above, a data element could be the distance to the ball. In `AiboCommIN`, data elements are either the received data from `AiboCommOUT`, or a score (with or without confidence levels as described in chapter 6.2.3).

If more complex data has to be transmitted, the serializer can be easily extended to serialize complex structs as well. The current implementation of AiboCommOpenR only supports the struct in listing 6.1.

Listing 6.1: OTransportVectorData struct

```
#ifndef _DATATYPES_H_
#define _DATATYPES_H_

#include <vector>
#include <string>

using std::vector;
using std::string;

struct OTransportVectorData {

    int actionID;
    vector<string> data;

    void AddData(string dataToAdd) {
        data.push_back(dataToAdd);
    }
};

#endif //_DATATYPES_H_
```

Currently messages are limited to 512 bytes. For simple data such as positions of objects, and discussing actions this seems enough. Because data is transmitted over wifi, too much data should not be transmitted anyway.

6.2 Design Tradeoffs

For the design outlined above, certain tradeoffs had to be made. For instance, because of using UDP and wireless LAN, packets may get lost and therefore the system should not stall when not receiving any replies from other AIBOs. The system should not depend on the distribution of data to work. Because of this more or less unreliable transmission medium, we want the AIBOs to work stand-alone and not being controlled by a leader process or leader dog. Since data is distributed amongst all the AIBOs, this is not necessary anyway. All AIBOs can make decisions on their own, using their own expert system.

The sent and received data is merged into a summary of the team's knowledge about the current state. This state is referred to as the global world. Each dog has its own, local, copy of the global world. Looking at the soccer example for instance, this world contains information about the position of the ball, and position data of other AIBOs that was received earlier. That data might be out-dated and is updated when new data is received. For the sake of simplicity, no multi-hop is used and therefore AIBOs that aren't within each others reach won't receive each others data. However, this can be added later, in combination with history and confidence levels.

6.2.1 Using Multi-Hop

Multi-hop can be added to the system when the field that the AIBOs are operating in is larger then the reach of AIBO's wireless LAN, or when a lot of obstacles are around which can block the communication. For instance, if AIBO were to operate within a house with walls, using multi-hop would give an advantage as seen in figure 6.4. Using multi-hop in the soccer example would not give any advantage, since the playing field is small enough for the wireless signal to get to all AIBOs in the team, and there are no large obstacles on the field.

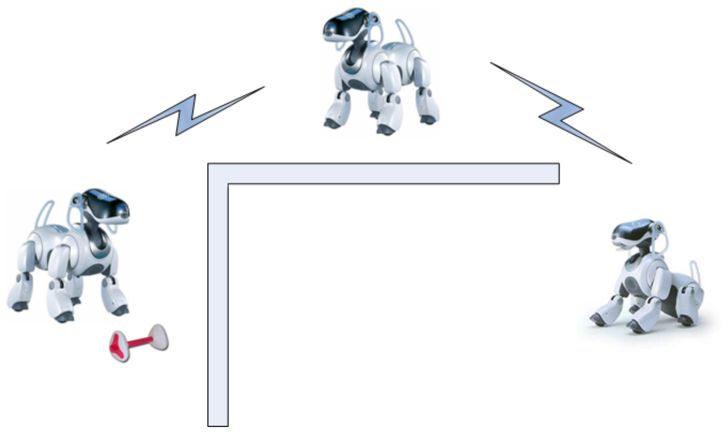


Figure 6.4: Using multi-hop

6.2.2 Keeping History

History can be kept from the global world that was discussed above, to be able to make assumptions for the future. The history of the ball position (i.e. in the soccer game) for instance, can give the direction that the ball is heading. This also applies to the positions of the other AIBOs. Using their directions and the direction of the ball, AIBO

©2008 Delft University of Technology

can decide whether to attack and proceed in the direction of the ball, or stay put and defend.

In the example of AIBO as a watchdog, history of the received data can be used to determine whether to explore a certain room or head the other direction if other AIBOs have been there before. A condition for this to work, however, is that the history needs to be *shared* and *distributed*. This can result in consistency problems. We need not only keep one AIBO's data up-to-date, but the data shared between other AIBOs as well. This can give problems in case of failure of an AIBO, or when an AIBO leaves the network for battery charging purposes.

It may also pay off to save the results of the *reasoning process* for future use. Saving these results in combination with their outcome (i.e. whether the action was successful) can give an idea whether similar actions in the future will be successful as well. Broadcasts for help can be kept to a minimum this way, and AIBO will turn into a learning agent.

The main problem with saving history is the fact that most data of the global world is outdated after a small period of time and therefore not usable anymore. Different applications also have different needs for saving history, and this history times out after different periods. For instance, ball positions in the soccer game are useless after a shorter period of time then information about rooms or walls in the watchdog example. Dealing with different applications therefore need different implementations of the history module. Therefore, history is not yet used in AiboCommOpenR.

6.2.3 Confidence Levels

Confidence levels can be used if it turns out that information from certain agents in the field is not reliable or accurate. They can be used either at the sending or at the receiving side, or even at both sides. If used at the sending side, they represent the confidence that AIBO has on the score that it sends back. A low confidence can be given if AIBO is unsure about some of his own data (e.g. his own position). This way the receiving AIBO can consider not using the received advise. A high confidence value, however, means that the receiving AIBO should use the advice.

If confidence levels are used at the receiving side, they represent the trust that the receiving AIBO has in the other AIBOs. In this situation, AIBO has an overview of all the agents and their corresponding trust values. This way AIBO can determine whether a received advise or score from a specific agent is reliable, and make a decision based on that. This trust value is calculated by the receiving AIBO itself, and is adjusted every time AIBO decides to use or not use a specific advise from an specific agent. Here a combination of both confidence levels can be used. Not using a score with a low confidence level can result in a higher trust value of the sending agent. The sending agent was not confident with his score and that results in a higher trust level for that agent. This is mathematically described in:

$$trust = clamp(0, 1, trust + \delta_p * (\gamma * trust * conf)) \tag{6.1}$$

where

$$\delta_p = \begin{cases} +1 & \text{if solution } s = \text{action } a; \\ -1 & \text{if solution } s \neq \text{action } a; \end{cases} \tag{6.2}$$

and *trust* represents the trust value of an AIBO, *conf* represents the confidence the sending AIBO has in his advise, γ is the trust learning rate, and $clamp(0, 1, trust, c)$ ensures that the value of c always lies in $(0, 1]$.[5][8]

This way an AIBO which proposes an incorrect advise with a high confidence value is penalized more heavily than one that proposes an incorrect advise but with a lower confidence value. Using confidence levels this way can result in a more reliable reasoning process, but also result in extra overhead, which is not desirable in this project because of the scarce resources of AIBO.

©2008 Delft University of Technology

©2008 Delft University of Technology

Expert Systems Overview 7

In order to let AIBO reason with the available data and the different actions that it can perform, a reasoning engine (expert system) is needed. Because such engines are widely available, only the two most important ones will be discussed in this chapter.

Knowledge-based expert systems, or simply expert systems, use human knowledge to solve problems that normally would require human intelligence. These expert systems represent the expertise knowledge as data or rules within the computer. These rules and data can be called upon when needed to solve problems.

Conventional computer programs perform tasks using conventional decision-making logic. The only knowledge they contain is the basic algorithm for solving a specific problem and the necessary boundary conditions. This program knowledge is often embedded as part of the programming code, so that as the knowledge changes, the program has to be changed and then rebuilt. Knowledge-based systems collect the small fragments of human know-how into a knowledge-base which is used to reason through a problem, using the knowledge that is appropriate. A different problem, within the domain of the knowledge-base, can be solved using the same program without reprogramming. The ability of these system to explain the reasoning process through back-traces, and to handle levels of confidence and uncertainty, provides an additional feature that conventional programming don't handle.[19]

7.1 CLIPS: C Language Integrated Production System

CLIPS was developed for NASA by the Software Technology Branch and the University of Georgia. It is available to the public and has several features which make it a desirable choice for producing expert systems. It offers the ability to integrate expert systems developed in CLIPS into the environment of a standard C/C++ program. This allows for quick prototyping and testing of expert systems alongside the development of a control program written in C/C++. An object-oriented programming language is also provided within CLIPS, called the CLIPS Object-Oriented Language (COOL).[20]

CLIPS uses forward-chaining. Its an empty tool and has to be filled with knowledge. After that, it can develop a solution, starting from the facts. For this, CLIPS uses a LISP-like procedural language.

CLIPS has the advantage of having an interpreted mode. This mode can be used for prototyping the agents before including them in the entire program. It is also useful for checking syntax, and for trying several ideas about the way in which rules should be written. This feature is very useful for fast development of several agents simultaneously.

Knowledge in CLIPS consists of ‘facts’, elementary information items:[21]

Ordered fact:

```
(person-name Franz J. Kurfess)
```

Deftemplate fact:

```
(deftemplate person ‘‘deftemplate example’’
    (slot name)
    (slot age)
    (slot eye-color)
    (slot hair-color)
)
```

To be able to reason with these facts, CLIPS needs ‘rules’:

General format:

```
(defrule <rule name> [‘‘comment’’]
    <patterns>* ; left-hand side (LHS)
                ; or antecedent of the rule
=>
    <actions>*  ; right-hand side (RHS)
                ; or consequent of the rule
)
```

Example rule:[21]

```
(defrule birthday-FJK
    (person (name ‘‘Franz J. Kurfess’’)
        (age 46)
        (eye-color brown)
        (hair-color brown)
    )
    (date-today April-13-02)
=>
    (printout t ‘‘Happy birthday, Franz!’’)
    (modify 1 (age 47))
)
```

More complex rule:[21]

```
(defrule silly-eye-hair-match
    (person (name ?name1)
        (eye-color ?eyes1&blue|green)
        (hair-color ?hair1&~black)
```

©2008 Delft University of Technology

```
        )
        (person (name ?name2&~?name1)
            (eye-color ?eyes2&~eyes1)
            (hair-color ?hair2&red|hair1)
        )
    =>
        (printout t ?name1 `` has ''?eyes1 `` eyes and '' ?hair1 `` hair.'')
        (printout t ?name2 `` has ''?eyes2 `` eyes and '' ?hair2 `` hair.'')
    )
```

Where:

```
    & = AND
    | = OR
    ~ = NOT
```

7.1.1 Limitations of CLIPS

CLIPS has its limitations as well. In other expert systems like 'LOOPS', rule sets can be arranged in a hierarchy, embedding one rule set inside another, etc. In CLIPS, there's only a single level of rule sets, which make things less clear. Also, there's only loose coupling of rules and objects, and rules cannot easily be embedded in objects, as in for example Centaur. Further more, CLIPS has no explicit agenda mechanism and only forward chaining is available as basic control flow. If an other kind of reasoning is to be implemented, tokens in working memory have to be manipulated.[21]

But despite these limitations, CLIPS is a good choice as the expert building tool to use in this project. Especially since it can be used in conjunction with the other tools (OPEN-R SDK) that are used.

7.1.2 Alternatives to CLIPS

As stated above, a lot more expert systems are available such as JESS, Eclipse and NEXPERT OBJECT. Also, there are various extensions and variants (like FuzzyCLIPS, AGENT CLIPS, DYNACLIPS, KnowExec, CAPE, PerlCLIPS, wxCLIPS and EHSIS) for CLIPS, that give it an advantage with respect to support of methods like fuzzy logic and agents.

Eclipse uses the same syntax as CLIPS (both are based on ART) and supports goal-driven (i.e. backwards) reasoning, a feature that's missing in CLIPS. Other features are a truth maintenance facility for checking consistency, and the ability to be integrated with C++ and dBase. With the help of a new extension 'RETE++', C++ header files can be generated for doing this integration. NEXPERT OBJECT uses another rule- and object-based system. It has facilities for designing graphical interfaces, and has a 'script language' for designing a user front-end. This, however, is of no use in this project.

JESS (Java Expert System Shell) is the other widely used expert system and is based on CLIPS. It uses the same syntax and has a large majority of the features of CLIPS. Instead of C/C++, it has a tight integration with Java and can be invoked easily from within Java applications. It can also utilize the object-oriented aspects of Java and as in CLIPS, it is also possible to call (Java) procedures, or be called by a procedure itself. JESS is still in development and more and more features are added as new versions of JESS are released.

7.2 CLIPS vs JESS

Since JESS is based on CLIPS, the syntax of both expert systems is roughly the same. The most important difference is the language it is based on. JESS is easily embedded in Java programs where CLIPS can be easily embedded in C/C++ applications. Other (minor) differences exist as well.[22]

CLIPS
The CLIPS shell provides the basic elements of an expert system:

1. a fact-list and instance-list: this is global memory for data
2. a knowledge-base: which contains all the rules
3. an inference engine: for controlling the execution of rules.

Facts are data that usually designate relations or information like `(is-animal dog)` or `(animals dog cat cow chicken)`. Rules can be applied on these facts in the form of IF-THEN rules. These rules are special because they 'fire' only once. Variables and wildcards can be used in the rules, and functions can be defined to manage the rules.

Besides these basic properties, CLIPS has pattern matching abilities (the Rete Algorithm), extended math functions, conditional tests, object-oriented programming (COOL: Clips Object-Oriented Language) with abstraction, inheritance, encapsulation, polymorphism and dynamic binding.[22]

JESS
JESS has the same basic properties as CLIPS, but its language lacks a few advanced features, for instance:

1. The CLIPS-user has a choice of seven strategies in rule-firing, while JESS only has the two most important ones.
2. The AND and OR conditional tests are not available in JESS in the left-hand side of rules.
3. JESS does not implement modules.
4. COOL: JESS lacks most COOL functions, but this is not really necessary since JESS is able to manipulate and directly reason about Java objects. Considering the

Java language is already object oriented, the lack of COOL is only a disadvantage when CLIPS code has to be ported to JESS.

In general, these shortcomings are not really an obstruction. The most harmful shortcomings of JESS are however situated in the performance of Java, and the fact that it is still work in progress:

1. it is generally accepted that JESS is up to three times (depending on the application) slower than CLIPS. This is primarily due to Java itself, although the implementation of JESS might also contribute significantly.

2. The JESS parser is considered less robust than CLIPS. The main reason for this is that, unlike CLIPS, JESS is still work in progress. However, the quality of JESS is improving with every new version.

3. As a consequence of the absence of some features, CLIPS code is sometimes hard (or even impossible) to port to JESS. This will certainly be the case in bigger projects.

On the other side, JESS has some advantages over CLIPS as well:

1. The Java language implementation makes JESS the choice for developing web-based expert systems, although some people say they prefer to use CLIPS with C++ in a CGI script. If Java is preferred, however, JESS is a more efficient choice.

2. JESS enables the user to put multiple expert systems in one process. Java threads can be used to run these in parallel.

To summarize, CLIPS is still more complete and stable than JESS, but this might change in the future, since the JESS package is being improved constantly. Besides that, JESS also has the property of using Java, which in the long run might prove to be a big advantage over CLIPS.

7.3 Implementing CLIPS

The choice between JESS and CLIPS depends on the application. If it is web-based or should reside in applet-form, the choice of JESS is a very logical one (which is even supported by the authors of CLIPS). For applications written in C++, CLIPS seems to be the most logical choice. It is therefore used in the reasoning engine of AiboCommOpenR.

AiboCommOpenR uses a special embedded version of CLIPS. For this a `CLIPSOutputFunction` has to be defined in `AiboReasoning` as an output route for CLIPS. Listing B.1 and B.2 of appendix B show how this is done for this project. Using the defined `CLIPSOutputFunction`, output from CLIPS can be redirected to `AiboReasoning` by defining the right output in the CLIPS rules:

```
(defrule print_final_score ‘‘print the calculated score’’
    (score ?C_score)
    (actionID ?C_actionID)
=>
    (printout t ‘‘Final score for acionID ’’ ?C_actionID ‘‘ is ’’ ?C_score)
    (printout ReturnToAiboReasoning ?C_actionID ?C_score)
)
```

Listing B.3 of appendix B shows how facts can be added to the CLIPS engine from within AiboCommOpenR.

©2008 Delft University of Technology

Scenarios 8

The technique outlined in chapter 6 can be used in numerous applications. Every application has its own strategies and therefore needs his own set of rules. Exploring a house needs a complete different strategy then being a pet dog for example. This chapter describes a few applications for AiboCommOpenR, and outlines different strategies where applicable.

8.1 AIBO as a Watchdog

Since AIBO has all sorts of input possibilities like video and audio, it can be used as a watchdog to monitor certain environments. It can register robberies and theft of items, and can even send a warning in case of fire or other suspicious events.

For this scenario, AIBO needs to walk around in a (possibly unknown) environment, investigating and analyzing the surroundings. This is like walking through a maze, where AiboCommOpenR can assist in sharing knowledge between different AIBOs about which rooms are already investigated, or help finding the way through the environment by discussing different routes. Multi-hop can be useful in this scenario in case of operating in large areas.

8.2 Using AIBO for Security and Surveillance

Besides using AIBO for keeping an eye on properties, it can also be used as a surveillance and security device for (elderly) people. In this application, AIBO doesn't register theft of devices or missing properties, but missing people or people having accidents like falling on the floor for example.

Also in this application, communication and collaboration is of great importance. It can be used for routing problems and share knowledge whether certain rooms are already investigated. The maze that was discussed earlier is applicable in this scenario as well. For instance if AIBO is used in unknown environments for search-and-rescue operations of people in buildings.

8.3 Using AIBO for Entertainment

AIBO can also be used as a pet dog or as a digital buddy. Digital buddies can use communication to play along with other AIBOs. This way AIBO can be a pet dog for elderly and lonely people, and become a social entity. Like in the previous application, the AIBOs can also use AiboCommOpenR to share knowledge. One example would be sharing the location of the energy station, in case one of the AIBOs is almost empty and can't find it himself.

8.4 AIBO in the 'Four-Legged Robot Soccer League'

The 'RoboCup Four-Legged League' is a robot soccer league where AIBO robots play in teams of four on a field of size 4m x 6m. In recent years the low level skills of the robots such as vision, localization, and ball handling have improved substantially. However, deliberate passing is extremely challenging and occurs rarely during games. If the AIBOs were able to communicate and play as a real team, it would contribute to a much more exciting game. For example, broadcasting ball information makes sure that all the players in the field have knowledge over the ball position, even when their view is blocked. This way none of the AIBOs get stalled searching for a ball that it can't see, and thus resulting in more action.

By sharing AIBO's plans, multiple AIBOs can cooperate and even play a one-two or use other passing techniques. Special defense and attack mechanism can be examined, and special formations can be tried, bringing the RoboCup soccer game to a higher level.

AiboCommOpenR can be used for the communication and reasoning part, and can therefore contribute to a better team play. Since this thesis focuses primarily on the use of AiboCommOpenR in the soccer league, a few strategies will be discussed next.

8.4.1 Strategies

It is desirable for the dogs to have specific roles and not to act only individually, but as a team. This way the dogs won't end up in a big mess that destroys every possibility to play nice soccer. The teamplay should be based upon the collective knowledge of all the dogs, which would clearly be an advantage compared to solitary AIBOs.

Using AiboCommOpenR for this purpose requires rules for the different kind of strategies that are used by the players that are in the RoboCup team. A goalkeeper should use a different kind of strategy than a defender for example.

The actions of the goalkeeper are mainly based on the fact that he must not end up too far from the own goal. This should be one of the first rules in the knowledge base. It is very important for the goalkeeper to know its own position with great accuracy. The goalkeeper is supposed to stay close to the goal line, turned towards the ball to facilitate savings and other appropriate actions by getting confident and fresh information through the his camera. In some situations it is possible that the ball, even though it is close to the goal, is not visible with the own camera. The knowledge about the ball position should then be based on information from the other AIBOs, if it is available (and trustworthy if confidence levels are used as described in chapter 6.2.3). To simplify this, the behavior of the goalkeeper can be made static in the sense that it should act the same way no matter how the field players are placed.

The field players differ from the goalkeeper by their dynamic behavior, and the fact that they move over larger areas. There are several tactical roles that an AIBO can have:[23]

- single defender

- single attacker
- left defender
- right defender
- left attacker
- right attacker.

Combinations of these roles depend on the number of active players and the chosen team positioning. Other roles that can be considered are supporting attacker and sweeper. The supporting attacker is supposed to be prepared to act as a second wave in the offence. The sweeper is supposed to defend the area closest to the own goal and just get the ball out of there. However, these two roles are only of interest if there are more than four players in a team.

Besides different roles, different positionings can be distinguished in the RoboCup game as well. Using the roles as stated above, the AIBOs can take a standard or a defensive positioning for example. A standard positioning has one defender, a left attacker, and a right attacker, as shown in figure 8.1.

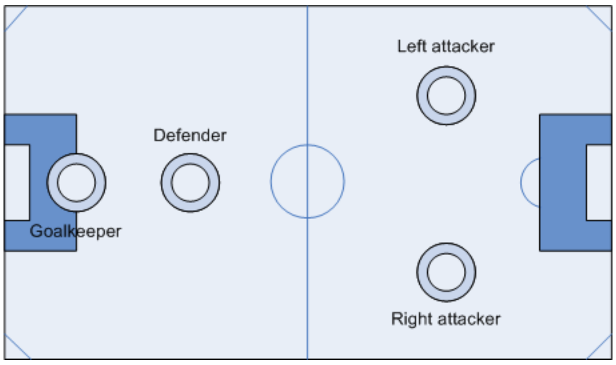


Figure 8.1: RoboCup standard positioning

The positioning has to change whenever a player is removed from the game, because of a penalty for example. The new disposition then consists of one defender and one attacker. Using AiboCommOpenR, the remaining AIBOs can reload their rule set for the new positioning, and get into an other role. This role can be reached in two different ways:[23]

- If an attacker is taken from the field, the remaining attacker gets the role of single attacker.
- If the defender is taken from the field, the attacker closest to the own goal gets the role single defender, and the other gets the role of single attacker.

An alternative positioning can be used when the focus lies on a strong defense. This can be suitable when the team is playing a very good team, the team is winning, or when the current result is satisfying for some reason. When a player is removed as a consequence of penalties, a change to an even stronger defense can be made. This disposition, consisting of two defenders, can also be reached in two ways:[23]

- If the attacker is taken from the field, nothing needs to be done.
- If any of the defenders are removed, the attacker is set to take this role.

Using these different strategies in combination with AiboCommOpenR can result in better teamplay, and a more exciting RoboCup soccer game.

Designing CLIPS Rules 9

As described in chapter 7.1, CLIPS needs rules in order to reason. Different kinds of rules are needed for the different kinds of applications outlined in chapter 8. This chapter will primarily focus on rules for the soccer game.

Currently, the AIBOs are already capable of playing soccer, so basic soccer rules are not necessary in the reasoning engine. However, they are still playing so called 'pupils football'. There is no communication what so ever, and all AIBOs play for themselves. For instance, if an AIBO can't find the ball because his view is blocked, he will keep looking for it, wandering around and thereby stalling the game. A simple sign from an other AIBO in the field could easily fix this problem.

Therefore, having some form of communication, and reasoning with data that AIBO receives from other players in the field can bring the soccer game to a higher level. Following are a few rules that can be used for the reasoning part of AiboCommOpenR, and their implementation in CLIPS.

9.1 Basic Soccer Game Rules

9.1.1 Moving Towards the Ball

The first rule uses the AIBO's own distance to the ball and the distance from the other AIBO to the ball, and calculates a score with that data for the idea of moving to the ball that the other AIBO has.

The basic facts needed for this rule are:

- AIBOs own the distance to the ball
- the distance from the other AIBO to the ball (has been send along with the data)
- the default score to start with, for calculating the final score.

```
(defrule calculate_score_using_distance ‘‘use distance’’
    (not (calc_using_distance done))
    (distance self ball ?D_me_ball)
    (distance other ball ?D_other_ball)
    ?sc <- (score ?C_score)
=>
    (printout t ‘‘Distance self-ball:  ’’  ?D_me_ball)
    (printout t ‘‘Distance other-ball:  ’’  ?D_other_ball)
    (printout t ‘‘Current score:  ’’  ?C_score)
```

```
        (if (< ?D_me_ball ?D_other_ball)
        then
            (printout t ``I'm closer to the ball!  decr score'')
            (retract ?sc)
            (assert (score (- ?C_score 1)))
        else
            (if (= ?D_me_ball ?D_other_ball)
            then
                (printout t ``Same distance to the ball, let him go!
                    incr score'')
                (retract ?sc)
                (assert (score (+ ?C_score 1)))
            else
                (printout t ``He is closer to the ball!  incr score'')
                (retract ?sc)
                (assert (score (+ ?C_score 1)))
            )
        )
        (assert (calc_using_distance done))
    )
```

Testing this rule with a default score of 5, a distance of AIBO 1 to the ball of 10, and a distance to the ball of AIBO 2 of 20 results in:

```
CLIPS> (run)
Distance self-ball:  10
Distance other-ball:  20
Current score:  5
I'm closer to the ball!  decr score
Final score is 4
CLIPS>
```

The score is correctly decreased by '1' since AIBO 1 (where CLIPS is running) is closer to the ball than AIBO 2 (the initiator). Therefore, AIBO 1 should be moving towards the ball instead of AIBO 2, and AIBO 1 should discourage AIBO 2 by returning a low score.

9.1.2 Goalkeeper Moving

The second rule can be used in combination with the previous rule. If the previous rule increased the score (i.e. it thought that moving was a good idea), the second rule has to check whether the initiating AIBO is not a goalkeeper. If it is, the score has to be decreased again if the goalkeeper is getting to far away from the goal.

The basic facts needed for this rule are:

- whether the initiating AIBO is a goalkeeper or not (has been send along with the data)
- the new distance between the goalkeeper and the goal (can be calculated using the old and new position that has been send along with the data)
- maximum distance between the goalkeeper and the goal
- the default score to start with, for calculating the final score.

```
(defrule calculate_score_moving_goalkeeper ``moving goalkeeper''
    (not (calc_moving_goalkeeper done))
    (role goalkeeper)
    (distance goal new ?D_goal_new)
    (distance goal max ?D_goal_max)
    ?sc <- (score ?C_score)
=>
    (printout t ``Goalkeepers new distance from the goal:  ''
        ?D_goal_new)
    (printout t ``Goalkeepers max distance from the goal:  ''
        ?D_goal_max)
    (printout t ``Current score:  ''  ?C_score)
    (if (> ?D_goal_new ?D_goal_max)
    then
        (printout t ``Goalkeeper reached goal_max!  decr score'' )
        (retract ?sc)
        ; decrease by 2 (really bad idea to be moving right now)
        (assert (score (- ?C_score 2)))
    )
    (assert (calc_moving_goalkeeper done))
)
```

Testing this rule in combination with the rule and basic facts from above, and using a role as goalkeeper with a distance from the goal of 6, and a maximum distance from the goal of 5 results in:

```
CLIPS> (run)
Goalkeepers new distance from the goal:  6
Goalkeepers max distance from the goal:  5
Current score:  5
Goalkeeper reached goal_max!  decr score
Distance self-ball:  10
Distance other-ball:  20
Current score:  3
I'm closer to the ball!  decr score
Final score is 2
```

```
CLIPS>
```

The score is decreased by ‘3’, because AIBO 1 is closer to the ball than AIBO 2 (and should move to the ball instead), and by moving, the goalkeeper (AIBO 2) would get to far away from the goal. AIBO 2 should therefore stay on his current position, and hence the low score.

Testing the rule in combination with the rule above, but with a distance from the ball for AIBO 1 of 20, and a distance from the ball for AIBO 2 of 10, shows that the score is decreased by ‘2’ first, and then increased by ‘1’ again. This is because when looking only at the distance between the ball and AIBO 2, moving towards it looks like a good idea. However, taking into account that AIBO 2 is a goalkeeper which gets to far away from the goal when moving towards the ball, AIBO 1 should return a low score for discouragement. It therefore decreases the score by ‘2’, resulting in a final score of ‘4’.

```
CLIPS> (run)
Goalkeepers new distance from the goal:  6
Goalkeepers max distance from the goal:  5
Current score:  5
Goalkeeper reached goal_max!  decr score
Distance self-ball:  20
Distance other-ball:  10
Current score:  3
He is closer to the ball!  incr score
Final score is 4
CLIPS>
```

9.1.3 Shooting the Ball

This rule uses the fact that AIBO 1 (where CLIPS is running) might see a different part of the play field than AIBO 2 (the initiator). This way, when AIBO 2 wants to shoot the ball to a specific part of the field, AIBO 1 can return a low score when it detects an enemy there, or a high score when a team player is close to that position.

For this rule, the field has to be split up in different numbered parts, which can then be used as directions for the ball, and for positions of other AIBOs. The part where the enemy-goal is in should have a special number, so that a high score can be returned when AIBO 2 is aiming at the goal and should therefore try to score.

The basic facts needed for this rule are:

- direction that the initiating AIBO wants to shoot the ball to (has been send along with the data)
- positions of enemy AIBOs

- positions of team players
- position of the enemy goal
- the default score to start with, for calculating the final score.

```
(defrule calculate_score_using_ball_direction "use ball direction"
    (not (calc_using_ball_direction done))
    (position ball new ?P_ball_new)
    (position self ?P_self)
    (position team_player_1 ?P_team_player_1)
    (position team_player_2 ?P_team_player_2)
    (position enemy_1 ?P_enemy_1)
    (position enemy_2 ?P_enemy_2)
    (position enemy_3 ?P_enemy_3)
    (position enemy_goal ?P_enemy_goal)
    ?sc <- (score ?C_score)
=>
    (printout t ``New ball position:  ''  ?P_ball_new)
    (printout t ``Self position:  ''  ?P_self)
    (printout t ``Team player 1 position:  ''  ?P_team_player_1)
    (printout t ``Team player 2 position:  ''  ?P_team_player_2)
    (printout t ``Enemy 1 position:  ''  ?P_enemy_1)
    (printout t ``Enemy 2 position:  ''  ?P_enemy_2)
    (printout t ``Enemy 3 position:  ''  ?P_enemy_3)
    (printout t ``Enemy goal position:  ''  ?P_enemy_goal)
    (printout t ``Current score:  " ?C_score)
    (if (= ?P_ball_new ?P_self)
    then
        (printout t ``Going to kick in my direction!  incr score'')
        (bind ?C_score (+ ?C_score 1))
    )
    (if (= ?P_ball_new ?P_team_player_1)
    then
        (printout t ``Going to kick in direction of team player 1!
            incr score'')
        (bind ?C_score (+ ?C_score 1))
    )
    (if (= ?P_ball_new ?P_team_player_2)
    then
        (printout t ``Going to kick in direction of team player 2!
            incr score'')
        (bind ?C_score (+ ?C_score 1))
    )
    (if (= ?P_ball_new ?P_enemy_1)
    then
```

```
            (printout t ``Going to kick in direction of enemy 1!
                decr score'')
            (bind ?C_score (- ?C_score 1))
        )
        (if (= ?P_ball_new ?P_enemy_2)
        then
            (printout t ``Going to kick in direction of enemy 2!
                decr score'')
            (bind ?C_score (- ?C_score 1))
        )
        (if (= ?P_ball_new ?P_enemy_3)
        then
            (printout t ``Going to kick in direction of enemy 3!
                decr score'')
            (bind ?C_score (- ?C_score 1))
        )
        (if (= ?P_ball_new ?P_enemy_goal)
        then
            (printout t ``Going to kick in direction of the goal!
                incr score'')
            ; increase with 2 (aiming at the goal is a good idea)
            (bind ?C_score (+ ?C_score 2))
        )
        (retract ?sc)
        (assert (score ?C_score))
        (assert (calc_using_ball_direction done))
    )
```

The rule increases the score whenever it finds a team player in the direction of the ball, and decreases it if there's an enemy there. If it finds more than one enemy, the score is decreased even more. This results in the following output when the rule is tested with two different positionings of the players:

```
CLIPS> (run)
New ball position:  4
Self position:  3
Team player 1 position:  3
Team player 2 position:  6
Enemy 1 position:  4
Enemy 2 position:  4
Enemy 3 position:  6
Enemy goal position:  10
Current score:  5
Going to kick in direction of enemy 1!  decr score
Going to kick in direction of enemy 2!  decr score
```

©2008 Delft University of Technology

```
Final score is 3
CLIPS>


CLIPS> (run)
New ball position:  4
Self position:  3
Team player 1 position:  4
Team player 2 position:  6
Enemy 1 position:  2
Enemy 2 position:  4
Enemy 3 position:  6
Enemy goal position:  10
Current score:  5
Going to kick in direction of team player 1!  incr score
Going to kick in direction of enemy 2!  decr score
Final score is 5
CLIPS>
```

The last example above shows that the score remains the same if there's a team player and an enemy in the direction that AIBO 2 wants to kick the ball to. AIBO 1 can't really give a good advice here, and therefore the score stays neutral.

If AIBO 2 is aiming at the goal, the following output is obtained:

```
CLIPS> (run)
New ball position:  10
Self position:  3
Team player 1 position:  3
Team player 2 position:  6
Enemy 1 position:  4
Enemy 2 position:  4
Enemy 3 position:  10
Enemy goal position:  10
Current score:  5
Going to kick in direction of enemy 3!  decr score
Going to kick in direction of the goal!  incr score
Final score is 6
CLIPS>
```

The rule adds '2' to the final score because AIBO 2 is aiming at the goal, but also decreases the score with '1' because an enemy is in the vicinity of the goal. Shooting the ball towards the goal while an enemy is near is not that smart, but worth a try. The final score is therefore a '6'.

©2008 Delft University of Technology

9.2 Remarks

The rules given above are simplified rules designed and implemented to be able to test AiboCommOpenR. If AiboCommOpenR is really to be used in the RoboCup soccer game, more complex rules are needed. As described in chapter 8.4.1, different positionings require different rule sets (i.e. focusing on a strong defense needs an other rule set then focusing on attacking). While the rules above don't implement that yet, they do already take into account the different roles that each player can have.

©2008 Delft University of Technology

Testing AiboCommOpenR 10

A proof of concept implementation of AiboCommOpenR has been made to be able to demonstrate the technique and to verify the design from chapter 6. A part of the source code of this implementation has been included as appendix B.

The proof of concept implementation has its own logging system which writes all the debug information to the memorystick that is used in AIBO. This chapter contains parts of these log files to demonstrate the flow of information between the various modules, when working on two different AIBOs. It uses the CLIPS rules from chapter 9.

10.1 Log Files

10.1.1 AiboDummyActionGenerator

`AiboDummyActionGenerator` waits for input from one of its sensors and then calls `DiscussActionS()`. In a real life situation, this module would be replaced by a module that benefits from reasoning with other AIBOs.

The log file shows that after initiating a discussion whether actionID 1313 is a good idea, it receives a 'go' from `AiboReasoning` (line 8 and 9), and then blinks its head-LEDs.

Listing 10.1: Log file of AiboDummyActionGenerator

```
1 AiboDummyActionGenerator.cc:AiboDummyActionGenerator:11@15324
      AiboDummyActionGenerator::AiboDummyActionGenerator()
2 AiboDummyActionGenerator.cc:DoInit:20@16994
      AiboDummyActionGenerator::DoInit()
3 AiboDummyActionGenerator.cc:DoInit:28@18981
      AiboDummyActionGenerator::DoInit() : OPENR::SetMotorPower(
      opowerON) 0
4 AiboDummyActionGenerator.cc:DoStart:36@19490
      AiboDummyActionGenerator::DoStart()
5 AiboDummyActionGenerator.cc:InitERS7SensorIndex:145@20595
      AiboDummyActionGenerator::InitERS7SensorIndex() : [31] PRM:/t2-
      Sensor:t2
6 AiboDummyActionGenerator.cc:InitERS7SensorIndex:145@21457
      AiboDummyActionGenerator::InitERS7SensorIndex() : [32] PRM:/t3-
      Sensor:t3
7 AiboDummyActionGenerator.cc:InitERS7SensorIndex:145@21979
      AiboDummyActionGenerator::InitERS7SensorIndex() : [33] PRM:/t4-
      Sensor:t4
8 AiboDummyActionGenerator.cc:DiscussActionS:71@27114
      AiboDummyActionGenerator::DiscussActionS() : going to discuss
      actionID 1313
9 AiboDummyActionGenerator.cc:PerformActionO:91@35467
      AiboDummyActionGenerator::PerformActionO() : doPerform action?
      TRUE
```

```
10 AiboDummyActionGenerator.cc:DoStop:51@88602
       AiboDummyActionGenerator::DoStop()
```

10.1.2 AiboReasoning

`AiboReasoning` starts by loading the CLIPS rules and then waits for a call to `DiscussActionO()`. The log shows that after receiving a call, `AiboReasoning` broadcasts this call to other AIBOs using `BroadcastActionS()`, and waits again (line 7 through 12). After a few milliseconds it receives data back from the other AIBOs (line 13 trough 15) and then uses these scores to determine what to send back to the initiating module (in this case `AiboDummyActionGenerator`). It decides to sends back a 'go' on line 21.

Line 22 through 32 show what happens if `AiboReasoning` receives a reasoning request from an other AIBO. It receives actionID 1234 and two data-elements from `AiboCommIN` (line 22 through 24), which are fed into CLIPS (line 25 through 27). The outcoming score is returned to AiboCommIN (line 30 and 31), which sends the score back to the AIBO it received the request from, using UDP.

Both sections of the log file show that the communication using the wifi network and UDP, and the reasoning part using CLIPS are working correctly.

Listing 10.2: Log file of AiboReasoning

```
1  AiboReasoning.cc:AiboReasoning:13@14670 AiboReasoning::
       AiboReasoning()
2  AiboReasoning.cc:DoInit:19@16994 AiboReasoning::DoInit()
3  AiboReasoning.cc:DoStart:29@19154 AiboReasoning::DoStart()
4  AiboReasoning.cc:DoStart:34@22499 AiboReasoning::DoStart() :
       loading CLIPS file
5  AiboReasoning.cc:DoStart:39@23323 AiboReasoning::DoStart() : file
       loaded
6  AiboReasoning.cc:DoStart:50@23665 AiboReasoning::DoStart() : CLIPS
        engine is now up and running
7  AiboReasoning.cc:DiscussActionO:102@27283 AiboReasoning::
       DiscussActionO() : going to discuss action for actionID 1313
8  AiboReasoning.cc:DiscussActionO:109@27538 AiboReasoning::
       DiscussActionO() : pendingActionToSend contains:
9  AiboReasoning.cc:DiscussActionO:110 OTransportVectorData ::
       actionID [1313] data—elements [test1, test2]
10 AiboReasoning.cc:BroadcastActionS:124@27878 AiboReasoning::
       BroadcastActionS() : going to broadcast for actionID 1313
11 AiboReasoning.cc:BroadcastActionS:125@28050 AiboReasoning::
       BroadcastActionS() : pendingActionToSend contains:
12 AiboReasoning.cc:BroadcastActionS:126 OTransportVectorData ::
       actionID [1313] data—elements [test1, test2]
13 AiboReasoning.cc:DonateScoresO:149@33426 AiboReasoning::
       DonateScoresO() : got 2 score(s) for actionID 1313
14 AiboReasoning.cc:DonateScoresO:150@33764 AiboReasoning::
       DonateScoresO() : receivedScores contains:
15 AiboReasoning.cc:DonateScoresO:151 OTransportVectorData ::
       actionID [1313] data—elements [10, 6]
16 AiboReasoning.cc:PerformActionS:170@34363 AiboReasoning::
       PerformActionS() : send back good advice...
```

```
17 AiboReasoning.cc:PerformActionS:173@34531 AiboReasoning::
       PerformActionS() : Reasoning advice for actionID 1313
18 AiboReasoning.cc:PerformActionS:180@34787 AiboReasoning::
       PerformActionS() : got '10' as a score
19 AiboReasoning.cc:PerformActionS:180@34955 AiboReasoning::
       PerformActionS() : got '6' as a score
20 AiboReasoning.cc:PerformActionS:185@35125 AiboReasoning::
       PerformActionS() : got '16' as a total score with 2 score(s)
       received
21 AiboReasoning.cc:PerformActionS:189@35294 AiboReasoning::
       PerformActionS() : returning advice: TRUE
22 AiboReasoning.cc:ReasonScoreO:215@62336 AiboReasoning::
       ReasonScoreO() : received reasoning request for actionID 1234
23 AiboReasoning.cc:ReasonScoreO:216@62592 AiboReasoning::
       ReasonScoreO() : receivedData contains:
24 AiboReasoning.cc:ReasonScoreO:217 OTransportVectorData :: actionID
        [1234] data-elements [77, 88]
25 AiboReasoning.cc:ReasonScoreO:230@62938 AiboReasoning::
       ReasonScoreO() : running command: '(assert (actionID 1234))'
26 AiboReasoning.cc:ReasonScoreO:236@63109 AiboReasoning::
       ReasonScoreO() : running command: '(assert (distance me ball
       10))'
27 AiboReasoning.cc:ReasonScoreO:242@63365 AiboReasoning::
       ReasonScoreO() : running command: '(assert (distance other ball
        77))'
28 AiboReasoning.cc:ReasonScoreO:247@63540 AiboReasoning::
       ReasonScoreO() : starting CLIPS reasoning
29 AiboReasoning.cc:CLIPSOutputFunction:296 AiboReasoning::
       CLIPSOutputFunction() : received from CLIPS : '1234::4'
30 AiboReasoning.cc:ReturnScoresS:259 AiboReasoning::ReturnScoresS() :
       got score from CLIPS : '1234::4'
31 AiboReasoning.cc:ReturnScoresS:260 AiboReasoning::ReturnScoresS() :
       returning score to AiboCommIN
32 AiboReasoning.cc:ReasonScoreO:251@63714 AiboReasoning::
       ReasonScoreO() : resetting CLIPS engine
33 AiboReasoning.cc:DoStop:73@91220 AiboReasoning::DoStop()
```

10.1.3 AiboCommOUT

The log file of `AiboCommOUT` shows that after receiving a broadcast request from `AiboReasoning` (line 8 through 13), it enables his `GivePulseO()` observer and waits for scores from other AIBOs (line 14 though 17). After a timeout, it donates the received scores to `AiboReasoning` on line 22 through 25.

Listing 10.3: Log file of AiboCommOUT

```
1 AiboCommOUT.cc:AiboCommOUT:18@13530 AiboCommOUT::AiboCommOUT()
2 AiboCommOUT.cc:DoInit:24@16994 AiboCommOUT::DoInit()
3 AiboCommOUT.cc:InitUDPBuffer:257@18066 AiboCommOUT::InitUDPBuffer
      ()
4 AiboCommOUT.cc:InitUDPBuffer:290@18377 AiboCommOUT::InitUDPBuffer
      () : success
5 AiboCommOUT.cc:DoStart:43@19154 AiboCommOUT::DoStart()
6 AiboCommOUT.cc:CreateUDPEndpoint:297@20166 AiboCommOUT::
      CreateUDPEndpoint()
7 AiboCommOUT.cc:Bind:317@20854 AiboCommOUT::Bind()
8 AiboCommOUT.cc:BroadcastActionO:343@28894 AiboCommOUT::
      BroadcastActionO() : going to broadcast serialized data '1313::
      test1::test2'
9 AiboCommOUT.cc:Send:108@29323 AiboCommOUT::Send()
```

```
AiboCommOUT.cc:SendCont:131@29494 AiboCommOUT::SendCont()
AiboCommOUT.cc:SendCont:145@29665 AiboCommOUT::sendData : 1313::test1::test2
AiboCommOUT.cc:SendCont:146@29835 AiboCommOUT::sendAddress : 0.0.0.0
AiboCommOUT.cc:SendCont:147@30008 AiboCommOUT::sendPort : 0
AiboCommOUT.cc:Receive:163@30179 AiboCommOUT::Receive()
AiboCommOUT.cc:ReceiveCont:180@30522 AiboCommOUT::ReceiveCont()
AiboCommOUT.cc:ReceiveCont:199@30692 AiboCommOUT::ReceiveCont : 7 bytes received
AiboCommOUT.cc:ReceiveCont:200@30946 AiboCommOUT::ReceiveCont : recvScore 1313::6
AiboCommOUT.cc:Close:214@31291 AiboCommOUT::Close()
AiboCommOUT.cc:Receive:163@31460 AiboCommOUT::Receive()
AiboCommOUT.cc:GivePulseO:403@31630 PulseGenerator::GivePulseO() : received pulse @ 31629
AiboCommOUT.cc:GivePulseO:408@31803 PulseGenerator::GivePulseO() : difference between start/end pulse: 1451ms
AiboCommOUT.cc:DonateScoresS:379@31973 AiboCommOUT::DonateScoresS() : received '10' as score for actionID 1313
AiboCommOUT.cc:DonateScoresS:379@32398 AiboCommOUT::DonateScoresS() : received '6' as score for actionID 1313
AiboCommOUT.cc:DonateScoresS:383@32570 AiboCommOUT::DonateScoresS() : donating vector:
AiboCommOUT.cc:DonateScoresS:384 OTransportVectorData :: actionID [1313] data-elements [10, 6]
AiboCommOUT.cc:CloseCont:232@33085 AiboCommOUT::CloseCont()
AiboCommOUT.cc:CreateUDPEndpoint:297@33427 AiboCommOUT::CreateUDPEndpoint()
AiboCommOUT.cc:Bind:317@33851 AiboCommOUT::Bind()
AiboCommOUT.cc:DoStop:70@88685 AiboCommOUT::DoStop()
```

10.1.4 AiboCommIN

`AiboCommIN` binds to a socket and waits for requests from other AIBOs. After receiving data it passes it along to `AiboReasoning` (line 9 through 12). After a few milliseconds it receives a score back from `AiboReasoning`, and it returns that score to the initiating AIBO on line 13.

Listing 10.4: Log file of AiboCommIN

```
1  AiboCommIN.cc:AiboCommIN:17@16602 AiboCommIN::AiboCommIN()
2  AiboCommIN.cc:DoInit:23@16994 AiboCommIN::DoInit()
3  AiboCommIN.cc:InitUDPBuffer:191@18067 AiboCommIN::InitUDPBuffer()
4  AiboCommIN.cc:InitUDPBuffer:210@18377 AiboCommIN::InitUDPBuffer() : success
5  AiboCommIN.cc:DoStart:42@19156 AiboCommIN::DoStart()
6  AiboCommIN.cc:CreateUDPEndpoint:217@20420 AiboCommIN::CreateUDPEndpoint()
7  AiboCommIN.cc:Bind:237@21280 AiboCommIN::Bind()
8  AiboCommIN.cc:Receive:101@21804 AiboCommIN::Receive()
9  AiboCommIN.cc:ReceiveCont:118@61312 AiboCommIN::ReceiveCont()
10 AiboCommIN.cc:ReceiveCont:136@61565 AiboCommIN::ReceiveCont : 12 bytes received
11 AiboCommIN.cc:ReceiveCont:137@61738 AiboCommIN::ReceiveCont : recvData 1234::77::88
12 AiboCommIN.cc:ReasonScoreS:263@61908 AiboCommIN::ReasonScoreS : passing data to AiboReasoning
13 AiboCommIN.cc:ReturnScoreO:281@63886 AiboCommIN::ReturnScoreO : returning score (1234::4) to initiating AIBO
```

```
AiboCommIN.cc:Receive:101@64059 AiboCommIN::Receive()
AiboCommIN.cc:DoStop:68@88348 AiboCommIN::DoStop()
```

10.1.5 PulseGenerator

Line 4 and 5 in the following log file show that the `PulseGenerator` starts sending pulses as soon as `AiboCommOUT` enables his corresponding `GivePulseO()` observer.

Listing 10.5: Log file of PulseGenerator

```
1 PulseGenerator.cc:PulseGenerator:8@15953 PulseGenerator::
     PulseGenerator()
2 PulseGenerator.cc:DoInit:14@16994 PulseGenerator::DoInit()
3 PulseGenerator.cc:DoStart:24@19155 PulseGenerator::DoStart()
4 PulseGenerator.cc:GivePulseS:53@30179 PulseGenerator::GivePulseS()
      : sending pulse
5 PulseGenerator.cc:GivePulseS:53@31973 PulseGenerator::GivePulseS()
      : sending pulse
6 PulseGenerator.cc:DoStop:35@88348 PulseGenerator::DoStop()
```

10.2 Results

Listing 10.2, 10.3 and 10.4 show that the whole information flow between the different modules is working as designed in chapter 6. The current proof of concept implementation demonstrates that AiboCommOpenR is capable of communicating and sharing knowledge (without history), and can use CLIPS on the AIBO for reasoning. It thereby shows that AiboCommOpenR can be used as a framework for collaborating AIBOs.

Listing 10.1 shows that the complete cycle of 'requesting a 'go' or 'no-go" and receiving one takes 35467 - 27114 = 8353ms, which is not acceptable in most of the applications described in chapter 8. However, disabling all the debug information (using the compiler-flag `-DOPENR_DEBUG` and `-DDODEBUG`) narrows this down to below 2 seconds. It seemed that writing the debug information to the memorystick, and writing the debug information to the console via wifi really slowed things down. Things can be speed up even more by lowering the wait timeout in `AiboCommOUT`. This way, AiboCommOpenR seems fast enough for the robots in the RoboCup game, and any of the other applications.

However, the current implementation cannot be used in combination with the currently available RoboCup software yet, simply because not all the rules for all the actions that AIBO has to perform in the RoboCup are designed yet, and there is no connection between the different modules of the RoboCup software and AiboCommOpenR at this moment. When these rules are available and a connection has been made, something about the usefulness of AiboCommOpenR in the RoboCup can be said. Currently the log files only show that the technique works as was designed in the previous chapters.

©2008 Delft University of Technology

Conclusion and Future Work 11

In this final chapter an overview will be presented of how the research objectives stated in the introduction of this thesis have been addressed. Some directions for future work will also be discussed.

11.1 Research Objectives

The main research objective, developing a collaborative framework for AIBO has been completed successfully. A design has been made and a fully working prototype has been implemented and tested. In this section the completion of the individual tasks leading to the realization of the main research objective are discussed.

Understanding of current collaborating techniques, by studying relevant literature.
A lot of research on the subject of collaborating agents has been done throughout the years. Chapter 2 contains a small summary of this research and concludes that currently available agent development frameworks can't be applied to AIBO, because of AIBO's limitations. However, the collaboration methods discussed in this chapter are applicable, and were used in the next chapters.

Evaluation of the different development platforms that are available for AIBO.
Since AIBO is a robot that is widely used in academic research, a lot of development platforms are available. In chapter 4 the most important ones were discussed, and it concludes that OPEN-R SDK is the platform of choice for this project. The decision was based on the fact that OPEN-R is fast, has the most advanced network capabilities, and the other SDKs were not yet fully complete at the time of starting with this project. After that, chapter 5 was written as a manual for OPEN-R object communication, which can be used by anyone wanting to write applications using the OPEN-R SDK.

Study of the expert systems that are currently available.
Expert systems are a popular research topic, and therefore expert systems are widely available. Two of the most important ones (CLIPS and JESS) were researched, and discussed in chapter 7. Because JESS is JAVA-based, and no Java Virtual Machine is available for AIBO, CLIPS turned out to be the right expert system for AiboComm-OpenR.

The format of CLIPS facts has been included in chapter 7.1, and was used for adding knowledge to the prototype.

Design of the software and corresponding communication protocol.
The design of the software and communication protocol for AiboCommOpenR was discussed in chapter 6.1 and 6.1.3. The corresponding class diagrams, sequence diagrams, and statechart diagrams are included in appendix A.

For the design, a few tradeoffs had to be made; whether the prototype should keep history, use confidence levels and/or use multi-hop. All off these were eventually not implemented, because of resource limitations, and/or were unnessecary for the RoboCup soccer game.

Implementation of the software in a development platform of choice.
The design from chapter 6.1 has been implemented in the OPEN-R SDK, and parts of the source code from the prototype implementation have been included in appendix B.

Certain test-modules were written as well, to be able to debug the program, and to emulate AIBO to be able to test the communication between different AIBOs, when only one AIBO was available.

Implementation of knowledge in an expert system.
A few test rules were designed and implemented in CLIPS, to be able to test AiboCommOpenR and the communication between the AIBOs. They were described in detail in chapter 9.

Evaluation of the software when running on AIBO.
Once the prototype implementation was complete and some CLIPS rules were implemented, AiboCommOpenR could be put to the test. It turned out that the design worked, and that the current proof of concept implementation is capable of communicating and sharing knowledge (without history), and can use CLIPS on the AIBO for reasoning.

However, the current implementation could not yet be tested in combination with the currently available RoboCup software, because not all the rules for all the actions that AIBO has to perform in the RoboCup were designed yet, and there was no connection between the different modules of the RoboCup software and AiboCommOpenR. When these items are resolved, something about the usefulness of AiboCommOpenR in the RoboCup soccer game can be said.

11.2 Future Work

As stated above, the current implementation is not complete yet. It shows that AIBO is able to communicate and share knowledge with other AIBOs, but can't be used in a real-life situation yet. It can however be used as a basis for further development for either the 'Four-Legged Robot Soccer League' or any of the other applications described in this thesis. For doing this, a connection with the software that currently operates the robot in other scenarios has to be made, and CLIPS rules for reasoning with the data for that specific application need to be designed.

As described in chapter 6.2 future implementations of AiboCommOpenR can also include history, from either the outcome of the reasoning process or from the received data, and can use confidence levels. Using history, the AIBO can leave the event-driven path that it uses now, and act more pro-actively. With history, AIBO can make assumptions for the future, calculating the direction that the ball is heading for example, and keep broadcasts to a minimum when using the outcome of previous broadcasts for the current situation.

Other enhancements of AiboCommOpenR could include sharing AIBOs plans, or actively broadcasting information about the environment. For the soccer game example, this could include information on the ball position. This way all players in the field have knowledge over the ball position, even when their view is blocked, leading to a more active game by not having all the AIBOs scanning the environment. By sharing AIBO's plans in the RoboCup league, multiple AIBOs can cooperate and even play a one-two or use other passing techniques. Furthermore, special defense and attack mechanism can be examined, and special formations can be tried. These enhancements put different constraints on the communication protocol, demanding communication to be more reliable. However, when implemented, they can bring the RoboCup game to a higher level, using collaboration.

©2008 Delft University of Technology

Bibliography

[1] Anonymous, "Multi-agent system," http://en.wikipedia.org/wiki/Multi-agent_system.

[2] K. P. Sycara, "Multi-agent systems," School of Computer Science at Carnegie Mellon University," Project report for American Association for Artificial Intelligence, Summer 1998.

[3] Stuart J. Russel, Peter Norvig, *Artificial Intelligence: A Modern Approach*, 2nd ed. Prentice Hall, 2003. [Online]. Available: http://aima.cs.berkeley.edu/

[4] Dr. Michael Schroeder, "Software Agents - Introduction," http://agents.soi.city.ac.uk/introslides/tsld001.htm, Based on: H. Nwana, Software Agents: An Overview, Knowledge Engineering Review, 1996, 11(3):205-244.

[5] E. Daman BSc., "Collaborative robot agents," Research Assignment, Delft University of Technology, 2005.

[6] e. a. Pedro Cuesta-Morales, Zahia Guessoum, "Agent technologies at work," http://upgrade-cepis.org/issues/2004/4/up5-4Presentation.pdf, August 2004.

[7] James Allen, Nate Blaylock, George Ferguson, "A problem solving model for collaborative agents," in *AAMAS '02: Proceedings of the first international joint conference on Autonomous agents and multiagent systems.* ACM Press, 2002, pp. 774–781. [Online]. Available: http://doi.acm.org/10.1145/544862.544923

[8] Yezdi Lashkari, Max Metral, Pattie Maes, "Collaborative Interface Agents," http://agents.www.media.mit.edu/groups/agents/publications/aaai-ymp/aaai.html.

[9] M.N. Huhns, M.P. Singh, "Distributed Artificial Intelligence for Information Systems," *CKBS-94 Tutorial*, June 1994.

[10] Fabio Bellifemine, Agostino Poggi, Giovanni Rimassa, "JADE - A FIPA-compliant agent framework," http://sharon.cselt.it/projects/jade/papers/PAAM.pdf.

[11] "JADE - Java Agent DEvelopment Framework," http://jade.tilab.com/.

[12] Anonymous, "Communications protocol," http://en.wikipedia.org/wiki/Communications_protocol.

[13] Gerard J. Holzmann, *Design and Validation of Computer Protocols.* Prentice Hall, 1991.

[14] Novell, Inc., "Novell's Networking Primer," http://www.novell.com/info/primer/prim05.html.

[15] "URBI," http://uei.ensta.fr/baillie/eng/urbi.html.

[16] "Tekkotsu Development Framework for AIBO Robots," http://www-2.cs.cmu.edu/~tekkotsu/.

[17] SONY Corporation, "[AIBO SDE] official web site," http://www.sony.net/openr/.

[18] SONY Corporation, "OPEN_R_SDK-docE-1.1.5-r1," http://openr.aibo.com.

[19] PC AI Magazine, "PC AI - Expert Systems," http://www.pcai.com/web/ai_info/expert_systems.html.

[20] R.E. Douglas, Jr., "Sneakers: A concurrent engineering demonstration system," Master's thesis, Worcester Polytechnic Institute, 1998.

[21] Dr. Franz J. Kurfess, "CPE/CSC 481: Knowledge-Based Systems," http://www.csc.calpoly.edu/~fkurfess/Courses/CSC-481/W03/Slides/2-CLIPS.ppt.

[22] K. V. Laerhoven, "Comparison of the clips and jess expert system shells," Computing Department at Lancaster University," Project report for Industrial Applications of AI, June 1999.

[23] e. a. Jakob Ahln, Jonas Eriksson, "Gifr main report," Department of Information Technology, Uppsala University," An undergraduate project in the course Project DV, fall 2003.

[24] Nicholas Roy, Gregory Dudek, *Collaborative Robot Exploration and Rendezvous.* Kluwer Academic Publishers, 2002. [Online]. Available: http://web.mit.edu/nickroy/www/papers/ARJournal.pdf

[25] Michael Wooldridge, Nick Jennings, "Intelligent Agents: Theory and Practice," *Knowledge Engineering Review*, vol. 10, no. 2, June 1995. [Online]. Available: http://www.csc.liv.ac.uk/~mjw/pubs/ker95/ker95.html

[26] M.R. Enesereth, S.P. Ketchpel, "Software Agents," *Communications of the ACMg*, vol. 37, no. 7, pp. 48–537, March 1994.

[27] Michael Weiss, "A gentle introduction to agents and their applications," http://www.magma.ca/~mrw/agents/.

[28] M.P. Papazoglou, S.C. Laufman, T.K. Sellis, "An organizational framework for cooperating intelligent information systems," *Journal of Intelligent and Cooperative Information Systems*, vol. 1, no. 1, pp. 169–202, 1992.

[29] Jean-Jack Riethoven, "Agents: the 'soft' kind," http://industry.ebi.ac.uk/~pow/Work/presentations/Agents/, March 2000.

[30] Tucker Balch, Lynne E. Parker, *Robot Teams: from diversity to polymorphism.* A.K. Peters, Ltd., 2002.

[31] Radia Perlman, *Interconnections: Bridges, Routers, Switches, and Internetworking Protocols.* Addison-Wesley, 1999.

©2008 Delft University of Technology

[32] Jeffrey S. Rosenschein, "An Introduction to MultiAgent Systems," `http://www.csc.liv.ac.uk/~mjw/pubs/imas/distrib/powerpoint-slides/`.

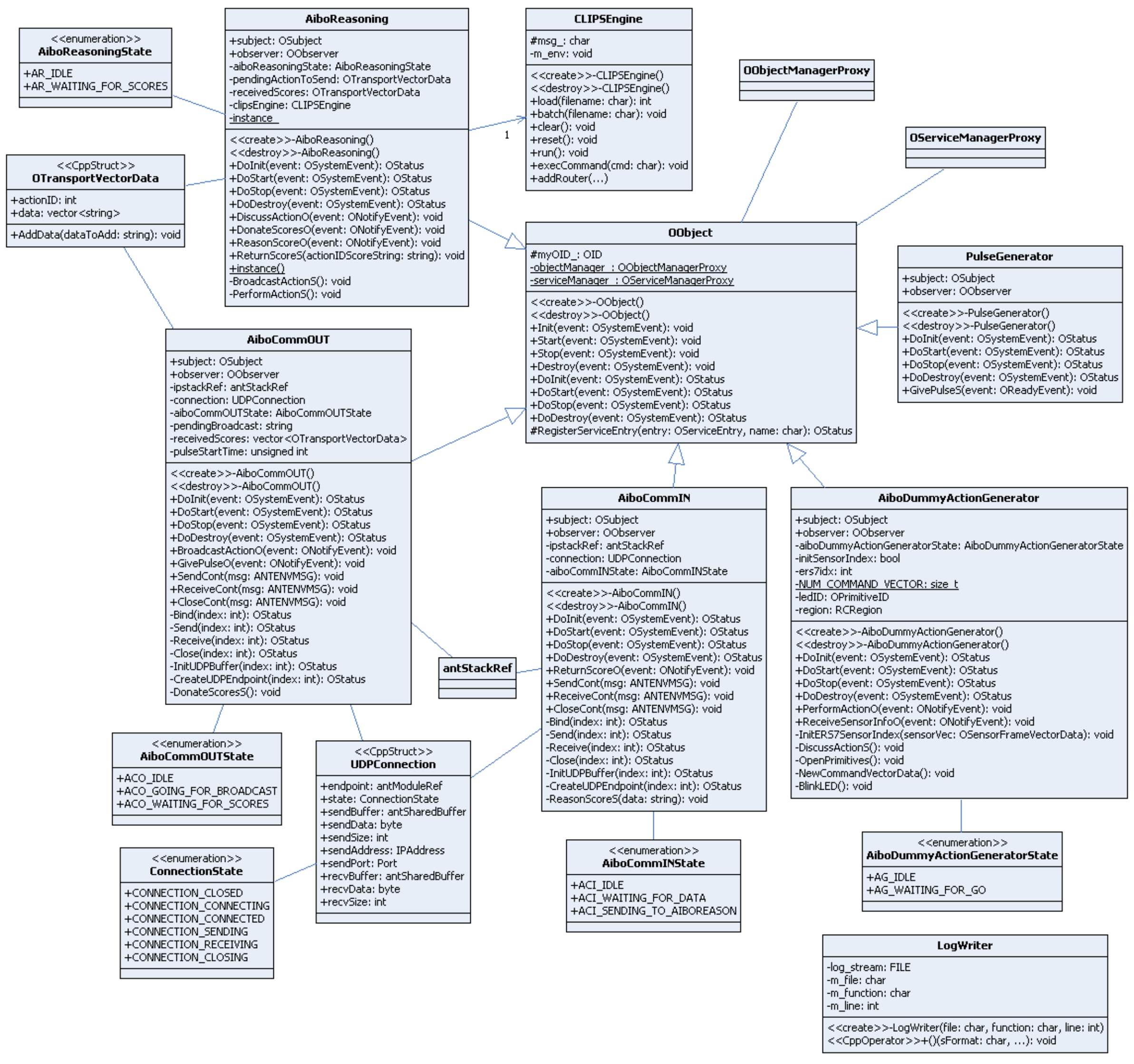
[33] Michael Wooldridge, *An Introduction to Multi-agent Systems*, 2nd ed. John Wiley & Sons Ltd, 2002.

©2008 Delft University of Technology

©2008 Delft University of Technology

A UML Diagrams

A.1 AiboCommOpenR Class Diagram



AiboReasoning
+subject: OSubject
+observer: OObserver
-aiboReasoningState: AiboReasoningState
-pendingActionToSend: OTransportVectorData
-receivedScores: OTransportVectorData
-clipsEngine: CLIPSEngine
-instance
<<create>>-AiboReasoning()
<<destroy>>-AiboReasoning()
+DoInit(event: OSystemEvent): OStatus
+DoStart(event: OSystemEvent): OStatus
+DoStop(event: OSystemEvent): OStatus
+DoDestroy(event: OSystemEvent): OStatus
+DiscussActionO(event: ONotifyEvent): void
+DonateScoresO(event: ONotifyEvent): void
+ReasonScoreO(event: ONotifyEvent): void
+ReturnScoreS(actionIDScoreString: string): void
+instance()
-BroadcastActionS(): void
-PerformActionS(): void
<<enumeration>>
AiboReasoningState
+AR_IDLE
+AR_WAITING_FOR_SCORES
<<CppStruct>>
OTransportVectorData
+actionID: int
+data: vector<string>
+AddData(dataToAdd: string): void
CLIPSEngine
#msg_: char
-m_env: void
<<create>>-CLIPSEngine()
<<destroy>>-CLIPSEngine()
+load(filename: char): int
+batch(filename: char): void
+clear(): void
+reset(): void
+run(): void
+execCommand(cmd: char): void
+addRouter(...)
1
OObjectManagerProxy
OServiceManagerProxy
OObject
#myOID_: OID
-objectManager_: OObjectManagerProxy
-serviceManager_: OServiceManagerProxy
<<create>>-OObject()
<<destroy>>-OObject()
+Init(event: OSystemEvent): void
+Start(event: OSystemEvent): void
+Stop(event: OSystemEvent): void
+Destroy(event: OSystemEvent): void
+DoInit(event: OSystemEvent): OStatus
+DoStart(event: OSystemEvent): OStatus
+DoStop(event: OSystemEvent): OStatus
+DoDestroy(event: OSystemEvent): OStatus
#RegisterServiceEntry(entry: OServiceEntry, name: char): OStatus
PulseGenerator
+subject: OSubject
+observer: OObserver
<<create>>-PulseGenerator()
<<destroy>>-PulseGenerator()
+DoInit(event: OSystemEvent): OStatus
+DoStart(event: OSystemEvent): OStatus
+DoStop(event: OSystemEvent): OStatus
+DoDestroy(event: OSystemEvent): OStatus
+GivePulseS(event: OReadyEvent): void
AiboCommOUT
+subject: OSubject
+observer: OObserver
-ipstackRef: antStackRef
-connection: UDPConnection
-aiboCommOUTState: AiboCommOUTState
-pendingBroadcast: string
-receivedScores: vector<OTransportVectorData>
-pulseStartTime: unsigned int
<<create>>-AiboCommOUT()
<<destroy>>-AiboCommOUT()
+DoInit(event: OSystemEvent): OStatus
+DoStart(event: OSystemEvent): OStatus
+DoStop(event: OSystemEvent): OStatus
+DoDestroy(event: OSystemEvent): OStatus
+BroadcastActionO(event: ONotifyEvent): void
+GivePulseO(event: ONotifyEvent): void
+SendCont(msg: ANTENVMSG): void
+ReceiveCont(msg: ANTENVMSG): void
+CloseCont(msg: ANTENVMSG): void
-Bind(index: int): OStatus
-Send(index: int): OStatus
-Receive(index: int): OStatus
-Close(index: int): OStatus
-InitUDPBuffer(index: int): OStatus
-CreateUDPEndpoint(index: int): OStatus
-DonateScoresS(): void
antStackRef
<<enumeration>>
AiboCommOUTState
+ACO_IDLE
+ACO_GOING_FOR_BROADCAST
+ACO_WAITING_FOR_SCORES
<<CppStruct>>
UDPConnection
+endpoint: antModuleRef
+state: ConnectionState
+sendBuffer: antSharedBuffer
+sendData: byte
+sendSize: int
+sendAddress: IPAddress
+sendPort: Port
+recvBuffer: antSharedBuffer
+recvData: byte
+recvSize: int
<<enumeration>>
ConnectionState
+CONNECTION_CLOSED
+CONNECTION_CONNECTING
+CONNECTION_CONNECTED
+CONNECTION_SENDING
+CONNECTION_RECEIVING
+CONNECTION_CLOSING
AiboCommIN
+subject: OSubject
+observer: OObserver
-ipstackRef: antStackRef
-connection: UDPConnection
-aiboCommINState: AiboCommINState
<<create>>-AiboCommIN()
<<destroy>>-AiboCommIN()
+DoInit(event: OSystemEvent): OStatus
+DoStart(event: OSystemEvent): OStatus
+DoStop(event: OSystemEvent): OStatus
+DoDestroy(event: OSystemEvent): OStatus
+ReturnScoreO(event: ONotifyEvent): void
+SendCont(msg: ANTENVMSG): void
+ReceiveCont(msg: ANTENVMSG): void
+CloseCont(msg: ANTENVMSG): void
-Bind(index: int): OStatus
-Send(index: int): OStatus
-Receive(index: int): OStatus
-Close(index: int): OStatus
-InitUDPBuffer(index: int): OStatus
-CreateUDPEndpoint(index: int): OStatus
-ReasonScoreS(data: string): void
<<enumeration>>
AiboCommINState
+ACI_IDLE
+ACI_WAITING_FOR_DATA
+ACI_SENDING_TO_AIBOREASON
AiboDummyActionGenerator
+subject: OSubject
+observer: OObserver
-aiboDummyActionGeneratorState: AiboDummyActionGeneratorState
-initSensorIndex: bool
-ers7idx: int
-NUM_COMMAND_VECTOR: size_t
-ledID: OPrimitiveID
-region: RCRegion
<<create>>-AiboDummyActionGenerator()
<<destroy>>-AiboDummyActionGenerator()
+DoInit(event: OSystemEvent): OStatus
+DoStart(event: OSystemEvent): OStatus
+DoStop(event: OSystemEvent): OStatus
+DoDestroy(event: OSystemEvent): OStatus
+PerformActionO(event: ONotifyEvent): void
+ReceiveSensorInfoO(event: ONotifyEvent): void
-InitERS7SensorIndex(sensorVec: OSensorFrameVectorData): void
-DiscussActionS(): void
-OpenPrimitives(): void
-NewCommandVectorData(): void
-BlinkLED(): void
<<enumeration>>
AiboDummyActionGeneratorState
+AG_IDLE
+AG_WAITING_FOR_GO
LogWriter
-log_stream: FILE
-m_file: char
-m_function: char
-m_line: int
<<create>>-LogWriter(file: char, function: char, line: int)
<<CppOperator>>+()(sFormat: char, ...): void

Figure A.1: AiboCommOpenR class diagram

A.2 AiboCommOpenR Sequence Diagrams

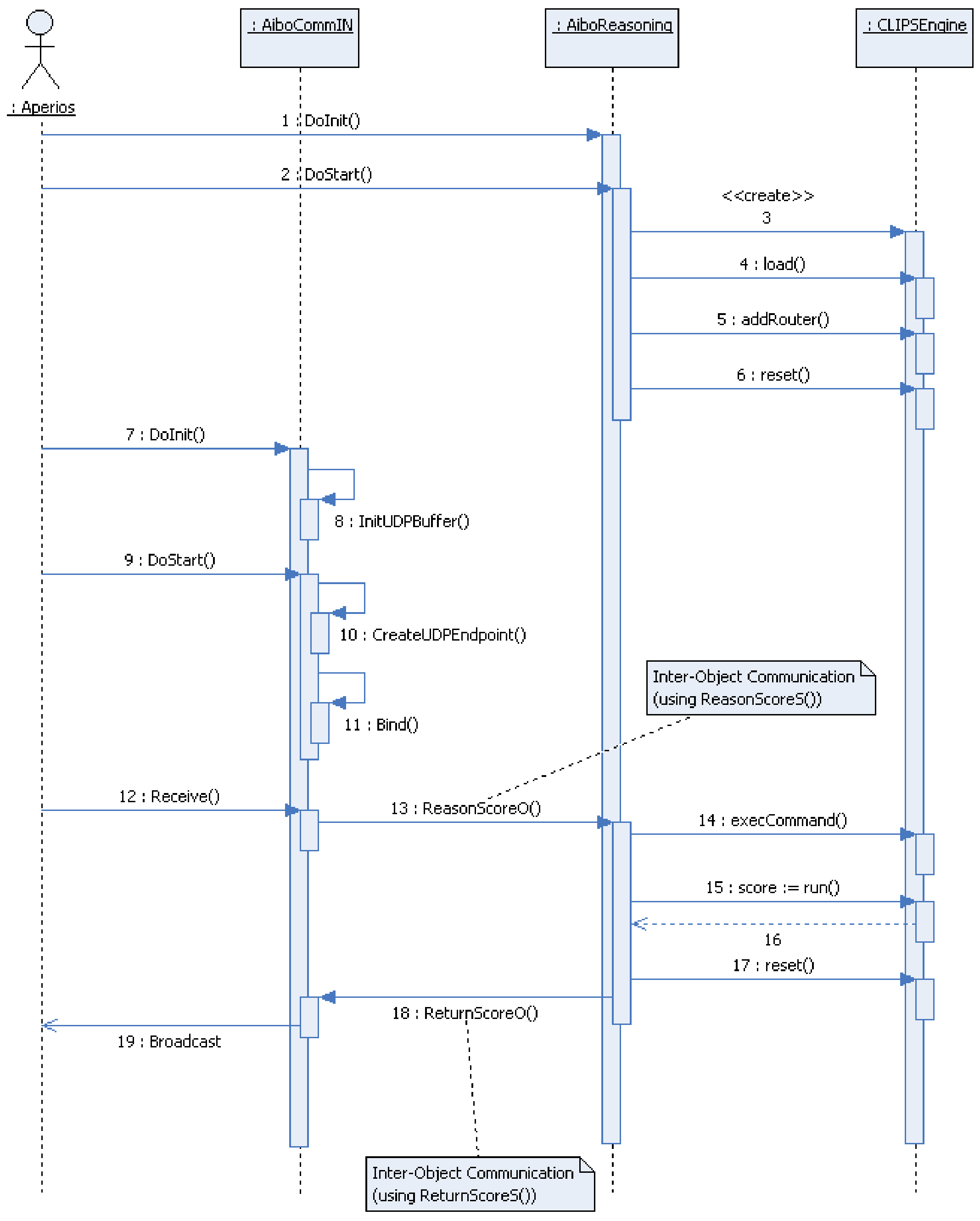


Figure A.2: AiboCommOpenR, AiboCommIN sequence diagram

©2008 Delft University of Technology

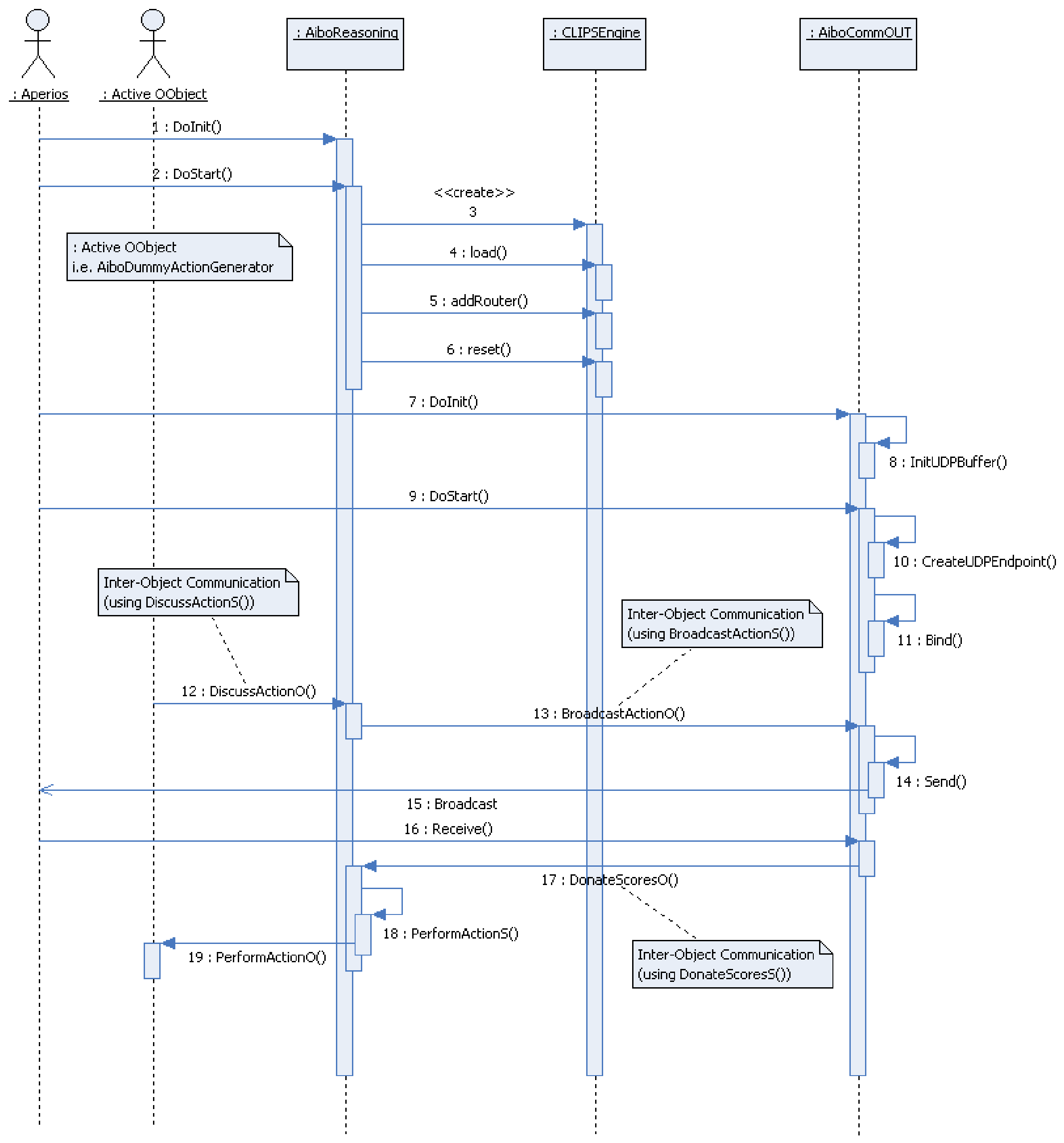


Figure A.3: AiboCommOpenR, AiboCommOUT sequence diagram

©2008 Delft University of Technology

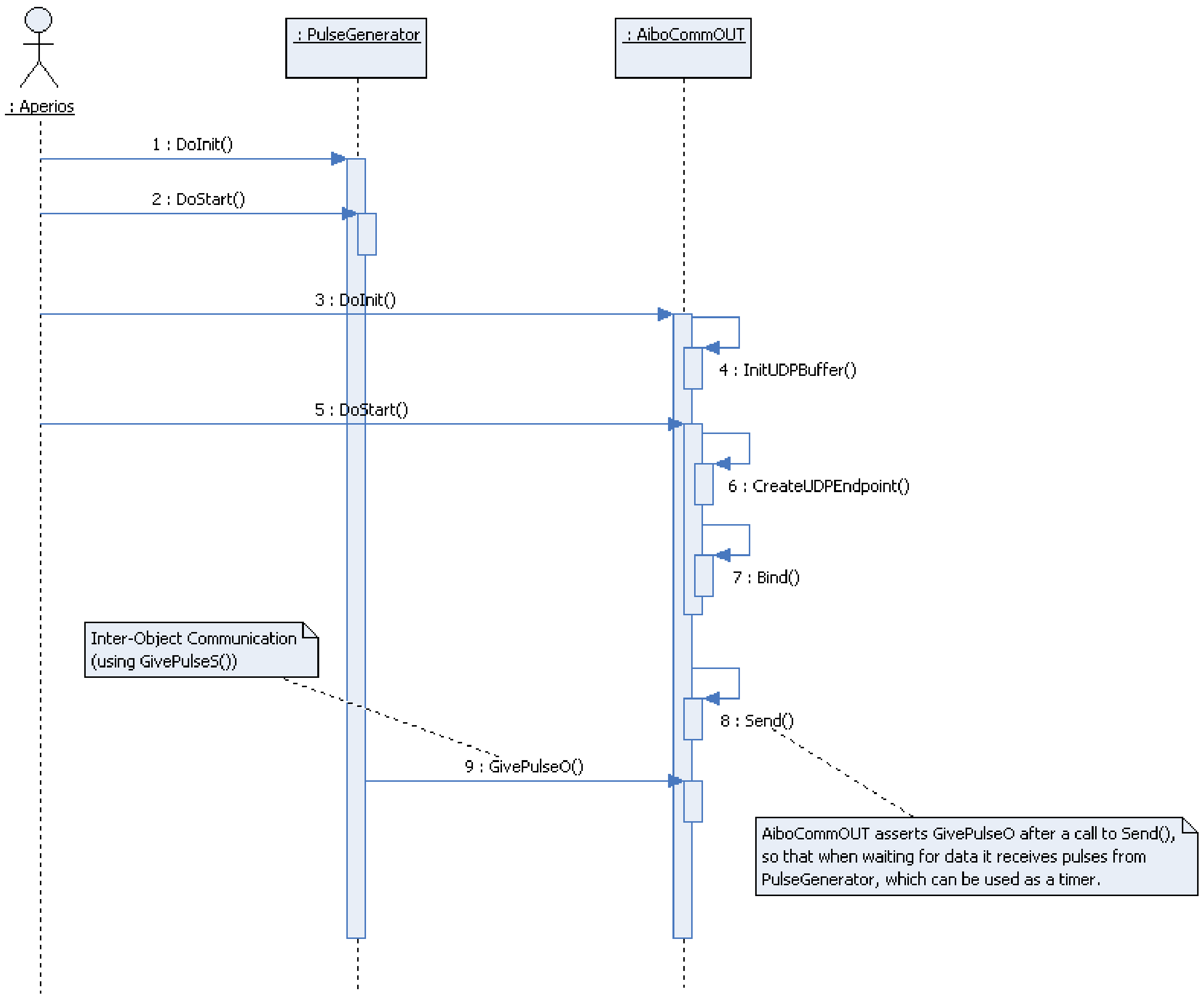

: Aperios
: PulseGenerator
: AiboCommOUT
1 : DoInit()
2 : DoStart()
3 : DoInit()
4 : InitUDPBuffer()
5 : DoStart()
6 : CreateUDPEndpoint()
7 : Bind()
Inter-Object Communication (using GivePulseS())
8 : Send()
9 : GivePulseO()
AiboCommOUT asserts GivePulseO after a call to Send(), so that when waiting for data it receives pulses from PulseGenerator, which can be used as a timer.

Figure A.4: AiboCommOpenR, PulseGenerator sequence diagram

©2008 Delft University of Technology

A.3 AiboCommOpenR Statechart Diagrams

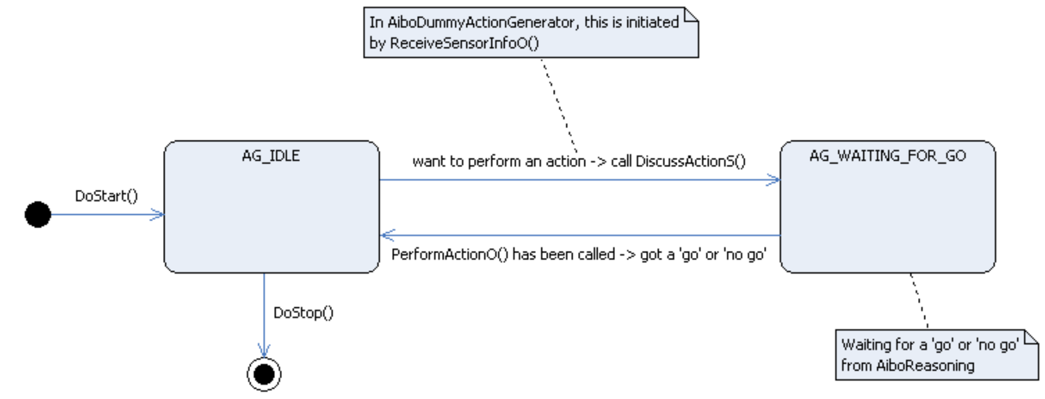

In AiboDummyActionGenerator, this is initiated by ReceiveSensorInfoO()
AG_IDLE
want to perform an action -> call DiscussActionS()
AG_WAITING_FOR_GO
DoStart()
PerformActionO() has been called -> got a 'go' or 'no go'
DoStop()
Waiting for a 'go' or 'no go' from AiboReasoning

Figure A.5: AiboCommOpenR, AiboDummyActionGenerator statechart diagram

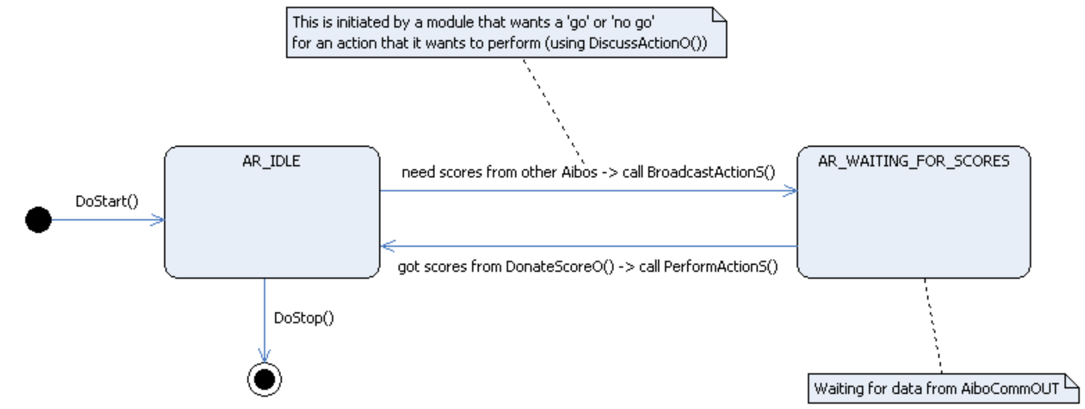

This is initiated by a module that wants a 'go' or 'no go' for an action that it wants to perform (using DiscussActionO())
AR_IDLE
need scores from other Aibos -> call BroadcastActionS()
AR_WAITING_FOR_SCORES
DoStart()
got scores from DonateScoreO() -> call PerformActionS()
DoStop()
Waiting for data from AiboCommOUT

Figure A.6: AiboCommOpenR, AiboReasoning statechart diagram

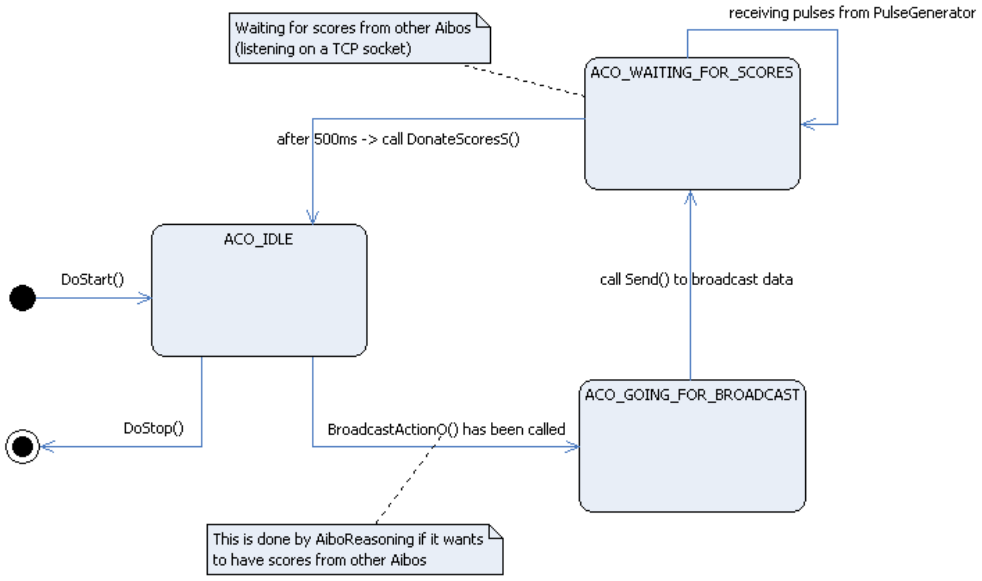

Waiting for scores from other Aibos (listening on a TCP socket)
receiving pulses from PulseGenerator
ACO_WAITING_FOR_SCORES
after 500ms -> call DonateScoresS()
ACO_IDLE
DoStart()
call Send() to broadcast data
ACO_GOING_FOR_BROADCAST
DoStop()
BroadcastActionO() has been called
This is done by AiboReasoning if it wants to have scores from other Aibos

Figure A.7: AiboCommOpenR, AiboCommOUT statechart diagram

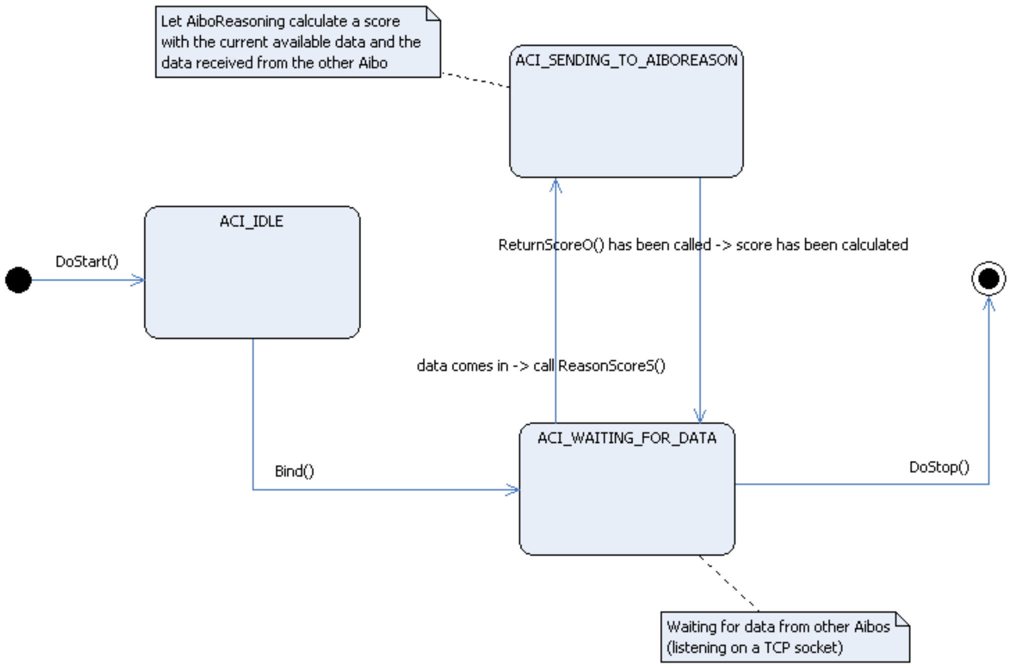

Let AiboReasoning calculate a score with the current available data and the data received from the other Aibo
ACI_SENDING_TO_AIBOREASON
ACI_IDLE
DoStart()
ReturnScoreO() has been called -> score has been calculated
data comes in -> call ReasonScoreS()
ACI_WAITING_FOR_DATA
Bind()
DoStop()
Waiting for data from other Aibos (listening on a TCP socket)

Figure A.8: AiboCommOpenR, AiboCommIN statechart diagram

Source Code B

AiboCommOpenR exists of numerous source code files and is therefore to large to be entirely

B.1 AiboReasoning

included in this appendix. However, a few important pieces of source code will be shown here.

Listing B.1: void AiboReasoning::DoStart(const OSystemEvent& event)

```
OStatus
AiboReasoning::DoStart(const OSystemEvent& event)
{
    DoDebug("AiboReasoning::DoStart()\n");

    // load the CLIPS reasoning-engine
    try {
        clipsEngine = new CLIPSEngine();
        DoDebug("AiboReasoning::DoStart() : loading CLIPS file\n")
            ;

        // load the default facts (contains all the rules for
            scores etc)
        int loadResult = clipsEngine->load("/MS/CONF/abreason.clp"
            );
        if (loadResult == 1) {
            DoDebug("AiboReasoning::DoStart() : file loaded\n");
        } else {
            throw CLIPSException("error loading CLIPS file");
        }

        // add a route to redirect CLIPS output to AiboReasoning
        clipsEngine->addRouter("ReturnToAiboReasoning",
            CLIPSQueryFunction, CLIPSOutputFunction);

        // and start the engine
        clipsEngine->reset();

        DoDebug("AiboReasoning::DoStart() : CLIPS engine is now up
             and running\n");
    } catch(CLIPSException *ex) {
        DoDebug("AiboReasoning::DoStart() : exception: %s\n", ex->
            why());
        delete ex;
    }

    // enable observers and subjects
    ENABLE_ALL_SUBJECT;
    aiboReasoningState = AR_IDLE;

    observer[obsDiscussActionO]->AssertReady();
    // and wait for reasoning-requests from AiboCommIN
    observer[obsReasonScoreO]->AssertReady();

    // and save a link to this instance (so CLIPS can return data)
    instance_ = this;

    return oSUCCESS;
}
```

Listing B.2: int CLIPSOutputFunction(void* env, char* logicalname, char* buffer)

```
int
CLIPSOutputFunction(void* env, char* logicalname, char* buffer)
{
    // CLIPS sends an actionID and score as two different buffers
    // append them and then return that string to AiboReasoning
    if (CLIPSNumReceived == 0) {
        CLIPSNumReceived++;
        strcat(CLIPSOutput, buffer);
        strcat(CLIPSOutput, "::");
    } else {
        strcat(CLIPSOutput, buffer);
        string toReturn = string(CLIPSOutput);

        OSYSDEBUG(("AiboReasoning::CLIPSOutputFunction() :
            received from CLIPS : '%s'\n", toReturn.c_str()));

        // reset for the next output from CLIPS
        CLIPSNumReceived = 0;
        strcpy(CLIPSOutput, "");

        // and then return the output of CLIPS to AiboCommIN
        AiboReasoning* ar = AiboReasoning::instance();
        ar->ReturnScoreS(toReturn);
    }
}
```

Listing B.3: void AiboReasoning::ReasonScoreO(const ONotifyEvent& event)

```
void
AiboReasoning::ReasonScoreO(const ONotifyEvent& event)
{
    // reasoning can be done in any state!

    // get the serialized OTransportVectorData (format: actionID::
        data::data::data)
    string scoreDataSerialized = (const char*)event.Data(0);

    // use al these data-elements as facts in CLIPS

    OTransportVectorData dataToReasonWith = unserialize(
        scoreDataSerialized);

    DoDebug("AiboReasoning::ReasonScoreO() : received reasoning
        request for actionID %d\n", dataToReasonWith.actionID);
    DoDebug("AiboReasoning::ReasonScoreO() : receivedData contains
        :\n");
    DoDebug_OTransportVectorData(&dataToReasonWith);

    // add the data as facts to the reasoning engine
    if (clipsEngine != 0) {

        // debug only -> change these into real-live values and
            add more facts
        int distance_me_ball = 10;
        // first element is the distance between the other aibo
            and the ball
        string distance_other_ball = (dataToReasonWith.data).at(0)
            ;

        // assert the actionID
        char command[64];
        sprintf(command, "(assert (actionID %d))",
            dataToReasonWith.actionID);
        DoDebug("AiboReasoning::ReasonScoreO() : running command:
            '%s'\n", command);
        clipsEngine->execCommand(command);

        // assert my own distance
        bzero(command, 64);
        sprintf(command, "(assert (distance me ball %d))",
            distance_me_ball);
        DoDebug("AiboReasoning::ReasonScoreO() : running command:
            '%s'\n", command);
        clipsEngine->execCommand(command);

        // assert the other aibo's distance
        bzero(command, 64);
        sprintf(command, "(assert (distance other ball %s))",
            distance_other_ball.c_str());
        DoDebug("AiboReasoning::ReasonScoreO() : running command:
            '%s'\n", command);
        clipsEngine->execCommand(command);
    }

    // and run CLIPS
    DoDebug("AiboReasoning::ReasonScoreO() : starting CLIPS
        reasoning\n");
    clipsEngine->run();

    // after running and receiving the output of CLIPS, reset
        everything for the next run
    DoDebug("AiboReasoning::ReasonScoreO() : resetting CLIPS
        engine\n");
    clipsEngine->reset();
}
```

Listing B.4: void AiboReasoning::ReturnScoreS(char* actionIDScoreString)

```
void
AiboReasoning::ReturnScoreS(string actionIDScoreString)
{
    // return score to AiboCommIN
    OSYSDEBUG(("AiboReasoning::ReturnScoreS() : got score from
        CLIPS : '%s'\n", actionIDScoreString.c_str()));
    OSYSDEBUG(("AiboReasoning::ReturnScoreS() : returning score to
        AiboCommIN\n"));

    // send the data (is already in the right actionID::score
        format)
```

©2008 Delft University of Technology

```
    subject[sbjReturnScoreS]->SetData(actionIDScoreString.c_str(),
        strlen(actionIDScoreString.c_str())+1);
    subject[sbjReturnScoreS]->NotifyObservers();

    // and wait for a new reasoning-request
    observer[obsReasonScoreO]->AssertReady();
}
```

Listing B.5: void AiboReasoning::DiscussActionO(const ONotifyEvent& event)

```
void
AiboReasoning::DiscussActionO(const ONotifyEvent& event)
{
    // only if we are idle (1 thing at the time. finish broadcast
        first)
    if (aiboReasoningState == AR_IDLE) {
        // received a request for discussion from an internal
            module. broadcast
        // this one and wait for scores

        // read actionID
        const int* actionID = (const int *)event.Data(0);
        DoDebug("AiboReasoning::DiscussActionO() : going to
            discuss action for actionID %d\n", *actionID);

        // make some test data to broadcast (for debugging
            purposes only)
        pendingActionToSend.actionID = *actionID;
        pendingActionToSend.AddData("test1");
        pendingActionToSend.AddData("test2");

        DoDebug("AiboReasoning::DiscussActionO() :
            pendingActionToSend contains:\n");
        DoDebug_OTransportVectorData(&pendingActionToSend);

        // and broadcast
        BroadcastActionS();
    } else {
        // do nothing until we are idle again
        DoDebug("AiboReasoning::DiscussActionO() (nothing to do
            here)\n");
    }
}
```

Listing B.6: void AiboReasoning::BroadcastActionS()

```
void
AiboReasoning::BroadcastActionS()
{
    // broadcast the pending action
    DoDebug("AiboReasoning::BroadcastActionS() : going to
        broadcast for actionID %d\n", pendingActionToSend.actionID)
        ;
    DoDebug("AiboReasoning::BroadcastActionS() :
        pendingActionToSend contains:\n");
    DoDebug_OTransportVectorData(&pendingActionToSend);

    // serialize and broadcast
    string pendingActionToSendSerialized = serialize(
        pendingActionToSend);
    subject[sbjBroadcastActionS]->SetData(
        pendingActionToSendSerialized.c_str(), strlen(
        pendingActionToSendSerialized.c_str())+1);
    subject[sbjBroadcastActionS]->NotifyObservers();

    // and wait for replies after broadcasting
    aiboReasoningState = AR_WAITING_FOR_SCORES;
    observer[obsDonateScoresO]->AssertReady();
}
```

Listing B.7: void AiboReasoning::DonateScoresO(const ONotifyEvent& event)

```
void
AiboReasoning::DonateScoresO(const ONotifyEvent& event)
{
    // if we are waiting for scores, accept them
    // (receiving from AiboCommOUT)
    if (aiboReasoningState == AR_WAITING_FOR_SCORES) {
        // read serialized OTransportVectorData
        string scoreDataSerialized = (const char*)event.Data(0);
        // save the received score in receivedScores
        receivedScores = unserialize(scoreDataSerialized);

        DoDebug("AiboReasoning::DonateScoresO() : got %d score(s)
            for actionID %d\n", (receivedScores.data).size(),
            receivedScores.actionID);
        DoDebug("AiboReasoning::DonateScoresO() : receivedScores
            contains:\n");
        DoDebug_OTransportVectorData(&receivedScores);

        // after receiving a score, the pending action is no
            longer pending
        pendingActionToSend.actionID = -1; pendingActionToSend.
            data.clear();

        // return a go or no-go
        PerformActionS();
    } else {
        // not waiting for scores -> do nothing
        DoDebug("AiboReasoning::DonateScoresO() (nothing to do
            here)\n");
    }
}
```

Listing B.8: void AiboReasoning::PerformActionS()

```
void
AiboReasoning::PerformActionS()
{
    // after receiving a score, we need to determine a go or no-go
    // (received score is in receivedScores as an
        OTransportVectorData element)

    DoDebug("AiboReasoning::PerformActionS() : send back good
        advice...\n");

    // simplicity: check if the average of the scores >= 5
    DoDebug("AiboReasoning::PerformActionS() : Reasoning advice
        for actionID %d\n", receivedScores.actionID);
    // get all scores in receivedScores
    int sumOfAllScores = 0; int totalNumberOfScores = 0;
    vector<string>::const_iterator i = (receivedScores.data).begin
        ();
    vector<string>::const_iterator end = (receivedScores.data).end
        ();
    while (i!=end) { // for each score-element (strings in the
        data element of receivedScores)
        // debug
        DoDebug("AiboReasoning::PerformActionS() : got '%d' as a
            score\n", atoi((*i).c_str()));
        sumOfAllScores = sumOfAllScores + atoi((*i).c_str());
        ++i;
        totalNumberOfScores++; // one more score
    }
    DoDebug("AiboReasoning::PerformActionS() : got '%d' as a total
         score with %d score(s) received\n", sumOfAllScores,
        totalNumberOfScores);

    bool adviceToReturn = ((sumOfAllScores / totalNumberOfScores)
        >= 5);

    DoDebug("AiboReasoning::PerformActionS() : returning advice: %
        s\n", adviceToReturn ? "TRUE" : "FALSE");

    subject[sbjPerformActionS]->SetData(&adviceToReturn, sizeof(
        adviceToReturn));
    subject[sbjPerformActionS]->NotifyObservers();

    // ready for a new action
    receivedScores.actionID = -1;
    // scores are no longer needed (no history)
    receivedScores.data.clear();
    // and ready for new requests
    aiboReasoningState = AR_IDLE;
    observer[obsDiscussActionO]->AssertReady();
}
```

B.2 AiboCommOUT

Listing B.9: void AiboCommOUT::BroadcastActionO(const ONotifyEvent& event)

```
void
AiboCommOUT::BroadcastActionO(const ONotifyEvent& event)
{
    // only if we are idle (because then we are binded)
    if (aiboCommOUTState == ACO_IDLE) {
        // received a request for broadcasting data from the
            AiboReasoning module, so send it around

        // read the data to broadcast
        string sendDataSerialized = (const char*)event.Data(0);
        DoDebug("AiboCommOUT::BroadcastActionO() : going to
            broadcast serialized data '%s'\n", sendDataSerialized.
            c_str());

        // save it in pendingBroadcast
        pendingBroadcast = sendDataSerialized;

        // for debugging only: add a score from myself already
        // this way we can continue if we are not receiving
            anything from other aibos
        OTransportVectorData tmpDummyScore;
        OTransportVectorData tmp = unserialize(pendingBroadcast);
        tmpDummyScore.actionID = tmp.actionID;
        tmpDummyScore.AddData("10");
        receivedScores.push_back(tmpDummyScore);

        // and broadcast
        aiboCommOUTState = ACO_GOING_FOR_BROADCAST;
        // transmit using the first available channel (always 0)
        Send(0);
    } else {
        // do nothing until we are idle again
        DoDebug("AiboCommOUT::BroadcastActionO() (nothing to do
            here)\n");
    }
}
```

Listing B.10: void AiboCommOUT::SendCont(ANTENVMSG msg)

```
void
AiboCommOUT::SendCont(ANTENVMSG msg)
{
    DoDebug("AiboCommOUT::SendCont()\n");

    UDPEndpointSendMsg* sendMsg = (UDPEndpointSendMsg*)antEnvMsg::
        Receive(msg);
    int index = (int)(sendMsg->continuation);
    if (connection[index].state == CONNECTION_CLOSED)
        return;

    if (sendMsg->error != UDP_SUCCESS) {
        OSYSLOG1((osyslogERROR, "%s : %s %d", "AiboCommOUT::
            SendCont()", "FAILED. sendMsg->error", sendMsg->error))
            ;
        DoDebug("%s : %s %d", "AiboCommOUT::SendCont()", "FAILED.
            sendMsg->error", sendMsg->error);
        Close(index);
        return;
    }
    char in_buff[64];
    DoDebug("AiboCommOUT::sendData : %s\n", connection[index].
        sendData);
    DoDebug("AiboCommOUT::sendAddress : %s\n", connection[index].
        sendAddress.GetAsString(in_buff));
    DoDebug("AiboCommOUT::sendPort : %d\n", connection[index].
        sendPort);

    // after the broadcast, start listening
    connection[index].state = CONNECTION_CONNECTED;
    connection[index].recvSize = SENDER_BUFFER_SIZE;
    // change sate
    aiboCommOUTState = ACO_WAITING_FOR_SCORES;
    // use the pulsegenerator as a timer
    pulseStartTime = getTime();
    observer[obsGivePulseO]->AssertReady();
    Receive(index);
}
```

Listing B.11: void AiboCommOUT::ReceiveCont(ANTENVMSG msg)

```
void
AiboCommOUT::ReceiveCont(ANTENVMSG msg)
{
    DoDebug("AiboCommOUT::ReceiveCont()\n");

    UDPEndpointReceiveMsg* receiveMsg = (UDPEndpointReceiveMsg*)
        antEnvMsg::Receive(msg);

    int index = (int)(receiveMsg->continuation);
    // only continue if we are waiting for scores
    if (connection[index].state == CONNECTION_CLOSED ||
        aiboCommOUTState != ACO_WAITING_FOR_SCORES) return;

    if (receiveMsg->error != UDP_SUCCESS) {
        OSYSLOG1((osyslogERROR, "%s : %s %d", "AiboCommOUT::
            ReceiveCont()", "FAILED. receiveMsg->error", receiveMsg
            ->error));
        DoDebug("%s : %s %d", "AiboCommOUT::ReceiveCont()", "
            FAILED. receiveMsg->error", receiveMsg->error);
        Close(index);
        return;
    }

    // determine the received string
    char recv_string[receiveMsg->size+1];
    memset(recv_string, '\0', sizeof(recv_string));
    memcpy(recv_string, receiveMsg->buffer, receiveMsg->size);
    DoDebug("AiboCommOUT::ReceiveCont : %d bytes received\n",
        receiveMsg->size);
    DoDebug("AiboCommOUT::ReceiveCont : recvScore %s\n",
        recv_string);
    // save the score in the receivedScore vector (as an
        OTransportVectorData element)
    // format of the received data: actionID::score
    receivedScores.push_back(unserialize(recv_string));

    // after receiving a score, we wait for more
    // the pulsegenerator is used for timing-out
    Close(index);
    Receive(index);
}
```

Listing B.12: void AiboCommOUT::DonateScoresS()

```
void
AiboCommOUT::DonateScoresS()
{
    // return the scores to AiboReasoning
    // scores are currently in receivedScores as seperate
        OTransportVectorData elements
    // convert them to one element
    vector<OTransportVectorData>::const_iterator i =
        receivedScores.begin();
    vector<OTransportVectorData>::const_iterator end =
        receivedScores.end();
    OTransportVectorData toDonate;
    // fetch the first actionID (are all the same)
    toDonate.actionID = (*i).actionID;
    while (i!=end) { //for each element
        // debug
        DoDebug("AiboCommOUT::DonateScoresS() : received '%s' as
            score for actionID %d\n", ((*i).data.back()).c_str(),
            toDonate.actionID);
        toDonate.AddData((*i).data.back()); // there's only 1
            element in each TransportVector's data-set
        ++i;
    }
    DoDebug("AiboCommOUT::DonateScoresS() : donating vector:\n");
    DoDebug_OTransportVectorData(&toDonate);

    // and return to AiboReasoning
    string stringToDonateSerialized = serialize(toDonate);
    subject[sbjDonateScoresS]->SetData(stringToDonateSerialized.
        c_str(), strlen(stringToDonateSerialized.c_str())+1);
    subject[sbjDonateScoresS]->NotifyObservers();
    // after returning the scores, we are able to broadcast again
    receivedScores.clear(); pendingBroadcast = "";
    aiboCommOUTState = ACO_IDLE;
    observer[obsBroadcastActionO]->AssertReady();
}
```

Listing B.13: void AiboCommOUT::GivePulseO(const ONotifyEvent& event)

```
void
AiboCommOUT::GivePulseO(const ONotifyEvent& event)
{
    // create a timer using the pulses from the PulseGenerator

    unsigned int curPulseTime = getTime();

    DoDebug("PulseGenerator::GivePulseO() : received pulse @ %d\n"
        , curPulseTime);

    // the time is in ms (since boot)
    // use a time-out of MAX_WAIT_TIME (default 500ms)
    if ((curPulseTime - pulseStartTime) > MAX_WAIT_TIME) {
        DoDebug("PulseGenerator::GivePulseO() : difference between
             start/end pulse: %dms\n", (curPulseTime -
            pulseStartTime));
        // disable all observers (not receiving input from other
            aibos anymore)
        DEASSERT_READY_TO_ALL_OBSERVER;
        // and return the received scores to AiboReasoning
        DonateScoresS();
    } else {
        // get the next pulse
        observer[obsGivePulseO]->AssertReady();
    }
}
```

B.3 AiboCommIN

Listing B.14: void AiboCommIN::ReceiveCont(ANTENVMSG msg)

```
void
AiboCommIN::ReceiveCont(ANTENVMSG msg)
```

```
{
    DoDebug("AiboCommIN::ReceiveCont()\n");

    UDPEndpointReceiveMsg* receiveMsg = (UDPEndpointReceiveMsg*)
        antEnvMsg::Receive(msg);

    int index = (int)(receiveMsg->continuation);
    if (connection[index].state == CONNECTION_CLOSED) return;

    if (receiveMsg->error != UDP_SUCCESS) {
        OSYSLOG1((osyslogERROR, "%s : %s %d", "AiboCommIN::
            ReceiveCont()", "FAILED. receiveMsg->error", receiveMsg
            ->error));
        DoDebug("%s : %s %d", "AiboCommIN::ReceiveCont()", "FAILED
            . receiveMsg->error", receiveMsg->error);
        Close(index);
        return;
    }

    // determine received string
    char recv_string[receiveMsg->size+1];
    memset(recv_string, '\0', sizeof(recv_string));
    memcpy(recv_string, receiveMsg->buffer, receiveMsg->size);
    DoDebug("AiboCommIN::ReceiveCont : %d bytes received\n",
        receiveMsg->size);
    DoDebug("AiboCommIN::ReceiveCont : recvData %s\n", recv_string
        );

    // format of the received data: actionID::data::data
    // send that to AiboReasoning (for determining a score)
    aiboCommINState = ACI_SENDING_TO_AIBOREASON;
    ReasonScoreS(string(recv_string));
}
```

Listing B.15: void AiboCommIN::ReasonScoreS(string data)

```
void
AiboCommIN::ReasonScoreS(string data)
{
    if (aiboCommINState == ACI_SENDING_TO_AIBOREASON) {
        // pass the data to AiboReasoning so it can determine a
            score
        DoDebug("AiboCommIN::ReasonScoreS : passing data to
            AiboReasoning\n");

        subject[sbjReasonScoreS]->SetData(data.c_str(), strlen(
            data.c_str())+1);
        subject[sbjReasonScoreS]->NotifyObservers();

        // after that, wait for a score from AiboReasoning
        observer[obsReturnScoreO]->AssertReady();
    } else {
        // not in send-mode so don't do anything (won't happen)
        DoDebug("AiboCommIN::ReasonScoreS() (nothing to do here)\n
            ");
    }
}
```

Listing B.16: void AiboCommIN::ReturnScoreO(const ONotifyEvent& event)

```
void
AiboCommIN::ReturnScoreO(const ONotifyEvent& event)
{
    string scoreDataSerialized = (const char*)event.Data(0);

    DoDebug("AiboCommIN::ReturnScoreO : returning score (%s) to
       initiating AIBO\n", scoreDataSerialized.c_str());

    // TODO implement return of score (send())

    // and wait for new requests from other aibos
    aiboCommINState = ACI_WAITING_FOR_DATA;
    Receive(0);
}
```

©2008 Delft University of Technology

©2008 Delft University of Technology

Paper C

This appendix contains a paper concerning collaborative robot agents, as submitted to the CGAMES 2008, November 3-5, 2008, Wolverhampton, UK. Acceptance of the submission is pending.

CGAMES is part of Game-On ® International Conference of The University of Wolverhampton, UK. It will be held in the Light House Media Centre, Wolverhampton, UK, and is the 13th annual conference of its kind. The main aim of the conference is to bring together researchers, games developers, sound, graphics, video, and animation developers, education and training industry, and students from around the world to exchange ideas on design methods, research and development, and programming techniques that are beneficial to the computer games industry and academia.

©2008 Delft University of Technology

Collaborative Robot Agents

E. Daman BSc., Z. Yang and L.J.M. Rothkrantz
Man-Machine Interaction Group
Delft University of Technology
2628CD Delft, The Netherlands
E-mail: L.J.M.Rothkrantz@tudelft.nl

***Abstract*—This paper describes how methods for collaborative agents can be applied to the AIBO (*A*rtificial *I*ntelligence ro*BO*t) of SONY Corporation. A method for collaboration will be explained in detail, and the design and implementation of a working prototype will be presented. The software works in an ad-hoc wifi network with clients that can connect and disconnect at all times. Data can be shared between the clients (AIBOs) so that they can determine the right objectives for themselves and fine-tune their actions with the others.**

***Index Terms*—Collaborative agents, problem solving model, AIBO, OPEN-R, CLIPS, AiboCommOpenR**

I. Introduction

IN computer science, a software agent is a piece of software that acts for an user or other program in a relationship of agency. Such 'acting on behalf of' implies the authority to decide which (and if) action is appropriate. The idea is that agents are not strictly invoked for a task, but activate themselves.

To date, agents have been successfully employed in multiple application endeavors, such as:

- data collection and filtering
- pattern recognition
- event notification
- data presentation
- planning and optimization
- rapid response implementation.

One might imagine an agent which is involved in all of these activities, sequencing through each of those states in response to environmental events, acting independently or in collaboration with a human client.

The number of application areas for software agents continues to expand. Currently, software agents are primarily used in research, but other areas such as education, commercial, law enforcement, intelligence, and even military are showing their interest as well, as future clients increasingly find themselves swamped by information overload. Agents can process vast volumes of data which would be unmanageable by human agents. Additionally, agents can analyze, assess, plan, and react to environmental events at super-human speeds.

A problem with designing agents is the fact that their competence is necessarily restricted to situations similar to those they have encountered in the past. The agents, when working on their own, have only a limited vision of the world, and have their resource limitations. If social ability is introduced however, and multiple agents are able to engage other components through some sort of communication and coordination, they may collaborate on a task, and collect information that would not be available to them if they were working alone. They could share knowledge and resources, which could lead to better decisions based on more information, and tasks may be completed in less time.

As with every advantage, sharing knowledge has its problems as well. The agents need to deal with issues like distributed or centralized data, the storage of data, and have to deal with multiple sources of information. Received and stored information may not be up-to-date anymore, and the original source of that information may be disconnected from the network of agents. Tradeoffs have to be made: is information distributed between all the agents, even to agents that can't see the original source? Maybe a centralized control center which keeps track of all the information and assigns tasks to the agents is more applicable? What to do with history of data, and do we trust the information that is received from other agents?

A. The Collaboration Problem

When several intelligent agents interact they may form a multi-agent system or multiple-agent system. The agents are considered to be autonomous entities collaborating with each other, and their interactions can be either cooperative or selfish. That is, the agents can share a common goal (e.g. an ant colony), or they can pursue their own interests (as in the free market economy).

Multi-agent systems can be used to solve problems which are difficult or impossible for an individual agent or monolithic system to solve. They have been successfully applied in domain areas like entertainment, computer games, air-traffic control, air combat, personal assistants, load-balancing, logistics, mining large (distributed) datasets in astronomy or transportation, finance, geographic information systems as well as in many other fields. Multi-agent systems are widely being advocated to be used in networking and mobile technologies, to achieve automatic and dynamic load balancing, high scalability, and self healing networks.

Central to the design and effective operation of collaborative agents are a core set of issues and research questions

E. Daman BSc. is a student of the Faculty of Electrical Engineering, Mathematics and Computer Science, Delft University of Technology, Mekelweg 4, 2628CD Delft, The Netherlands. E-mail: e.daman@student.tudelft.nl

©2008 Delft University of Technology

that have been studied over the years by the distributed AI community:[1]

- How do we formulate, describe, decompose, and allocate problems and synthesize results among a group of intelligent agents?
- How do we enable agents to communicate and interact? What communication languages and protocols do we use? What and when can they communicate? How can we find useful agents in an open environment?
- How do we ensure that agents act coherently in making decisions or taking action, accommodating the nonlocal effects of local decisions and avoiding harmful interactions? How do we ensure the agents do not become resource bounded? How do we avoid unstable system behavior?
- How do we enable individual agents to represent and reason about the actions, plans, and knowledge of other agents to coordinate with them. How do we reason about the state of their coordinated process (for example, initiation and completion)?
- How do we recognize and reconcile disparate viewpoints and conflicting intentions among a collection of agents trying to coordinate their actions?
- How do we engineer and constrain practical distributed AI systems? How do we design technology platforms and development methodologies for multi-agent systems?

This paper describes the design and development of a system of collaborative AIBO-robots, and specifically focuses on AIBO in the 'Four-Legged Robot Soccer League'. A collaboration method will be described in detail, and the design and implementation of a working prototype will be presented. The software works in an ad-hoc wifi network with clients that can connect and disconnect at all times. Data can be shared between the clients (AIBOs) so that they can determine the right objectives for themselves and fine-tune their actions with the others. The robots will stay autonomous, since the rules of the RoboCup soccer game clearly states that the use of a central controller PC is forbidden.

Fig. 1. Four-Legged Robot Soccer League

Currently, the AIBOs are already capable of playing soccer, however, they are still playing so called 'pupils football'. There is no communication what so ever, and all AIBOs play for themselves. For instance, if an AIBO can't find the ball because his view is blocked, he will keep looking for it, wandering around and thereby stalling the game. A simple sign from an other AIBO in the field could easily fix this problem.

Therefore, having some form of communication, and reasoning with data that AIBO receives from other players in the field can bring the soccer game to a higher level.

Section II first describes the different kind of agents that exist and the collaborative agent is discussed in further detail. It also includes a detailed problem solving model, that is used in section III. Section III describes the design and implementation of AiboCommOpenR, the software that is written for the system of collaborating AIBOs. The conclusion to this paper is presented in section IV.

II. Designing Collaborative Agents

A global definition of the term 'agent' would be 'a component of software and/or hardware which is perceiving its environment through sensors and acting on the environment through actuators, thus effecting what it senses in the future'.[2] The term percept is used to refer to the agent's perceptual inputs at any given instant. Perception is initiated by sensors and provides agents with information about the world they inhabit (see figure 2).

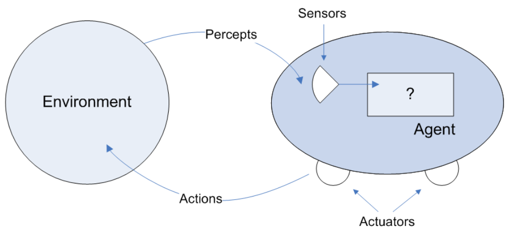


Fig. 2. Agent definition

This tends to obscure the differences between radically different approaches. Agents differ in the way they communicate and co-operate and so on. The different kind of agents can be split into 7 categories:[3][4]

- Collaborative agents
- Interface agents
- Mobile agents *(static vs. mobile)*
- Information/Internet agents
- Reactive agents *(deliberative vs. reactive)*
- Hybrid agents
- Smart agents.

All these type of agents have different motivations and challenges. This paper focuses on 'collaborative agents'. These agents negotiate in order to resolve conflicts and they collaborate to integrate information.

Collaborative agents need to be able to:[5]

- discuss and negotiate goals

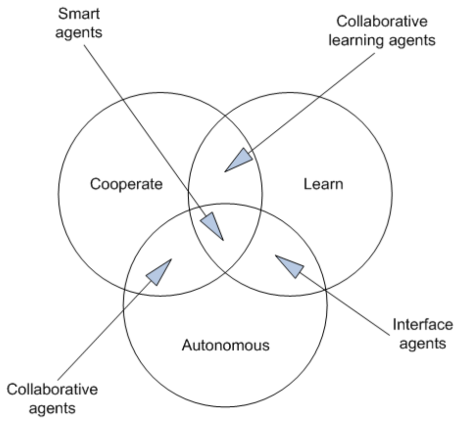


Fig. 3. Agent classification

- discuss options and decide on courses of action, including assigning different parts of a task to different agents
- discuss limitations and problems with the current course of action and negotiate modifications
- assess the current situation and explore future possibilities
- discuss and determine resource allocation
- discuss and negotiate initiative in the interactions
- perform parts of the task and report to others to update shared knowledge of the situation.

The main motivation factor for designing cooperative agents is the fact that some problems are simply to large for a centralized single agent. Resources might be limited or there is too much risk involved. But beside resources limitations, other motivations play an important role as well:[6]

- Allow for interconnection/interoperability of multiple existing legacy systems.
- Provide solutions to inherently distributed problems, e.g. distributed sensor networks or air-traffic control.
- Provide solutions which draw from distributed information sources.
- Provide solutions where the expertise is distributed, e.g. in health care provisioning.
- Enhance modularity (which reduces complexity), speed (due to parallelism), reliability (due to redundancy), flexibility (i.e. new tasks are composed more easily from the more modular organisation) and reusability at the knowledge level (hence shareability of resources).
- Research into other issues, e.g. understanding interactions among human societies.

So by letting agents cooperate we are able to:

> "create a system that interconnects separately developed collaborative agents, thus enabling the ensemble to function beyond the capabilities of any of its members"[7]

The central issue of collaborative problem solving is that in order to collaborate to achieve a common goal, the agents must coordinate their individual actions. Therefore, some sort of 'Collaborative Problem Solving Level' must exist, and the agents should capture the joint objectives, and the jointly chosen recipes and resources etc. to accomplish those objectives.

The collaborative problem solving level must serve 2 purposes. It must provide the structure that enables and drives interactions, and it should provide the connection between the joint intentions and the individual actions that an agent performs as part of the joint plan, while allowing an agent to have its own objectives. Agents must agree to some level of detail on the abstract joint recipes that they construct in order to collaborate. An agent does not need to know how the other agent accomplishes things below this level of abstraction and agents do not need to have the same library of recipes.[4]

Being able to solve problems, the cognitive state of the agent, affected by problem-solving actions, need to contain at least the following:[5]

- current situation
- intended objectives
- active objectives
- intended recipes
- recipe library
- function on objectives and situations giving recipes.

As a summary, the different phases to agent problem solving can be summed as follows:[5]

1) Determine the objective *(agents may reconsider objectives at any time).*
2) Determine the recipe *(find a recipe that may achieve the objective).*
3) Use the selected recipe and identify the next action to perform from the recipe *(if the recipe returns a sub-objective, go to step 1).*

A. Collaborative Framework

While a particular agent may not have any prior knowledge about a certain problem, there may exist a number of agents who do. Instead of each agent re-learning what other agents have already learned through experience, agents can simply ask for help. This gives each agent access to a potentially vast body of experience that already exists. There are two classes of such collaboration:[6]

- *Desperation based communication* which is invoked when a particular agent has insufficient experience to make a confident decision.
- *Exploratory communication* communication, on the other hand, is initiated by agents in bids to find the best set of peer agents to ask for help in certain classes of situations. Exploratory communication is undertaken by agents to discover new (as yet untried) agents who are better advisors in the solving of the agents' problems than the current set of peers they have previously tested.

Both forms of communication may occur at two orthogonal levels. At the *situation level*, desperation communication refers to an agent asking its peers for help in dealing with a new situation, while exploratory communication refers to an agent asking previously untested peers for how they would deal with old situations for which it knows the correct action, to determine whether these new agents are good advisors for future problems. At the *agent level*, desperation communication refers to an agent asking trusted peers to recommend an agent that its peers trust, while exploratory communication refers to agents asking peers for their evaluation of a particular agent perhaps to see how well these peers' modeling of a particular agent corresponds with their own.

Collaborative communication between agents occurs in the form of request and reply messages. An agent is not required to reply to any message it receives. This leaves each agent the freedom to decide when and whom to help.

Agents model peers' abilities to assist with solving problems by a trust value. For each class of situations an agent has a list of peers with associated trust values. Trust values vary between 0 and 1. The trust values reflect the degree to which an agent is willing to trust a peer's solution for a particular situation class. A trust value represents a probability that a peer's solution for a certain problem is a good solution based on prior history of proposed solutions from the peer. Trust values are further discussed in section III-C3.[4]

When an agent sends out a solution request to more than one peer it's likely to receive many replies, each with a potentially different solution and confidence value. In addition, the agent has a trust value associated with each peer. This gives rise to many possible strategies which an agent can use to choose a solution and a confidence value for this solution. Both trust and peer confidence should play a role in determining which proposed solution gets selected and with what confidence. Each proposed action is assigned a trust-confidence sum value which is the trust weighted sum of the confidence values of all the peers predicting this action. The action with the highest trust-confidence sum is chosen.

B. A Detailed Problem Solving Model

The following problem solving model is primarily focused on human-machine collaboration, but can be equally well applied to interactions between sophisticated software agents that need to coordinate their activities.[5]

An overview of the model is shown in figure 4. At the heart of the model is the *problem solving level*, which describes how a single agent solves problems. An agent might adopt an obligation or might evaluate the likelihood that a certain action will achieve that objective. This level is based on a fairly standard model of agent behavior. The problem solving level is specialized to a particular task and domain by a *task model*. The task model describes how to perform certain tasks, such as what possible objectives are, how objectives are (or might be) related, what resources are available, and how to perform specific problem solving actions such as evaluating a course of action. For an isolated autonomous agent, these two levels suffice to describe its behavior, including the planning and execution of task-level actions. For collaborative activity, more is needed.

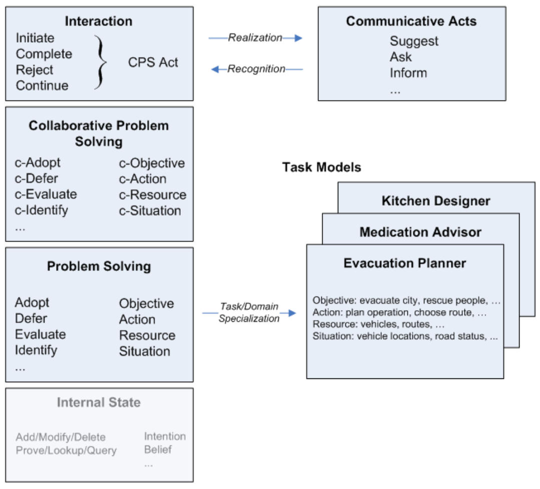


Fig. 4. Collaborative problem solving model

The *collaborative problem solving level* builds on the single-agent problem solving level. The collaborative problem solving actions parallel the single-agent ones, except that they are joint actions involving jointly understood objects. For example, the agents can jointly adopt an intention (making it a joint intention), or they can jointly identify a relevant resource, and so on. Finally, an agent cannot simply perform a collaborative action by itself. The *interaction level* consists of actions performed by individuals in order to perform their part of collaborative problem solving acts. Thus, for example, one agent may initiate a collaborative act to adopt a joint intention, and another may complete the collaborative act by agreeing to adopt the intention.

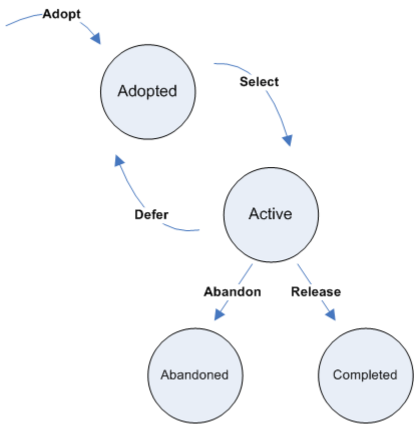


Fig. 5. Life cycle of an intention

When two agents collaborate to achieve goals, they must coordinate their individual actions. To mirror the development at the problem solving level, the collaborative problem solving level operates on the collaborative problem solving (CPS) state, which captures the joint objectives, the recipes jointly chosen to achieve those objectives, the resources jointly chosen for the recipes, and so on.[5]

The collaborative problem solving model must serve two critical purposes. First it must provide the structure that enables and drives the interactions between the agents as they decide on joint objectives, actions and behavior. In doing so, it provides the framework for intention recognition, and it provides the constraints that force agents to interact in ways that maintain the collaborative problem solving state. Second, it must provide the connection between the joint intentions and the individual actions that an agent performs as part of the joint plan, while still allowing an agent to have other individual objectives of its own.

Establishing part of the collaborative problem solving state requires an agreement between the agents. One agent will propose an objective, recipe, or resource, and the other can accept, reject or produce a counterproposal or request further information. This is the level that captures the agent interactions. To communicate, the agent receiving a message must be able to identify what CPS act was intended, and then generates responses that are appropriate to that intention.

III. Design and Implementation of AiboCommOpenR

Applying the problem solving model described in the previous section to AIBO results in a collaboration platform called AiboCommOpenR, which can be used for the collaboration between different AIBO-robots.

AiboCommOpenR is written in the OPEN-R SDK, which is a cross-development environment based on gcc (C++). OPEN-R is chosen because it has the most advanced networking capabilities, is the fastest in runtime, and the software that is written in it runs stand-alone on AIBO itself, without the use of a controller PC.

Developing software in OPEN-R deals with the complex process of inter-object communication, which is therefore further described in the next subsection.

A. *OPEN-R Objects*

In OPEN-R, multiple objects with various functionality are running concurrently and communicate with each other via inter-object communication (see figure 6). The concept of an object in OPEN-R is similar to one of a process in the UNIX or Windows operating systems.

An object is a concept that only exists at run-time. Each object has a counterpart in the form of an executable file, created at compile-time. Source code is compiled and linked (with the use of the OPEN-R SDK) to create this executable file. The file is put on an AIBO Programming Memory Stick and when AIBO boots, the operating system loads the file from the AIBO Programming Memory Stick and executes it as an object.[8]

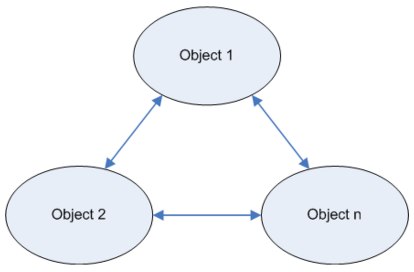


Fig. 6. OPEN-R Application software

Below is the typical life cycle of an object:

1) loaded by the system
2) wait for a message
3) when a message arrives, execute the method corresponding to the selector specified in the message and send some messages to other objects when necessary
4) when the method finishes execution, go to step 2.

This is an infinite loop. An object cannot terminate itself and persists while the system is activated.

OPEN-R objects communicate with each other while they perform their tasks. In OPEN-R, this communication between objects is called ‘inter-object communication’. The use of inter-object communication enables each object to be created separately and later be connected to other objects. When two objects communicate, the side that sends data is called the ‘subject’, and the side that receives data is called the ‘observer’. Figure 7 shows a case where the subject of object A communicates with the observer of object B.

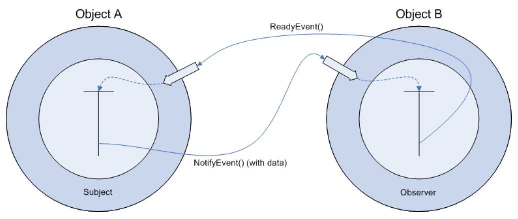


Fig. 7. Inter-object communication

When the observer is in a state ready to receive data, the observer sends `ASSERT-READY` to the subject. When the observer is in a state not ready to receive data, the observer sends `DEASSERT-READY` to the subject. The subject receives these in the form of a `ReadyEvent`. When the function `IsAssert()` in the `ReadyEvent` returns `true`, the subject can send a `NotifyEvent` to the observer. `NotifyEvent` includes the data that the subject wants to send to the observer.

Objects have entry points for receiving messages. As shown in figure 8, each entry point corresponds to a particular method of the object, and each method corresponds to a particular

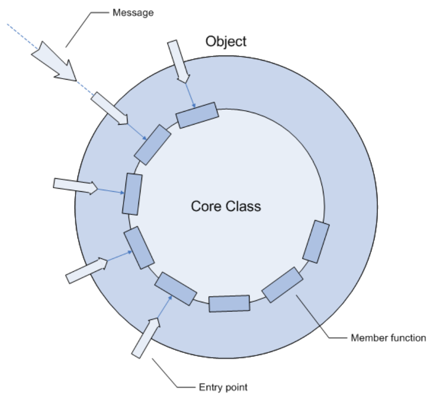


Fig. 8. A core class

member function of the object. In C++, an OPEN-R object can be represented by a core class, which has the following characteristics:[8]

- A core class inherits from the OObject class (which is the base class for all objects).
- A core class implements the `DoInit()`, `DoStart()`, `DoStop()` and `DoDestroy()` functions.
- A core class has the necessary number of OSubject and OObserver

Some member functions in the core class correspond to specific methods in the object:

1) Methods that are called at startup and shutdown:
   - *Init method*
     This is called at startup and initializes instances and variables.
   - *Start method*
     This is called at startup after Init is executed in all objects.
   - *Stop method*
     This is called at shutdown.
   - *Destroy method*
     This is called at shutdown after Stop is executed in all objects and destroys the subject and observer instances.

   The Init method, Start method, Stop method, and Destroy method correspond to each `DoInit()`, `DoStart()`, `DoStop()` and `DoDestroy()` function in the object's corresponding core class, respectively.
2) Methods that are called when a message is received from another object:
   Methods used in subjects:
   - *Control method*
     This receives the connection results between the subject and its observers.
   - *Ready method*
     The subject receives `ASSERT-READY` or `DEASSERT-READY` notifications from the observers.

   Methods used in observers:
   - *Connect method*
     This receives the connection results between an observer and its subjects.
   - *Notify method*
     This receives a message from the subject.

These inter-object communication techniques are needed for the design of AiboCommOpenR, which exists of several objects, each handling a different part of the collaboration framework.

B. Prototype Implementation

A number of issues have to be addressed in order to develop a successful distributed system for AIBO. These issues pertain to several different subject areas, such as a functioning motion system, which deals with mechanical robotics, as well as a functioning vision system, which deals with digital image analysis and color calibration. Furthermore, the system deals with artificial intelligence, distributed systems, real time systems and communication.

The focus for this paper has been on the distribution and communication issue. The main goal is the design and implementation of a system that runs on the AIBO and operates concurrently with the modules that are already developed (e.g. the motion modules for the soccer game). The system consists of several modules (OPEN-R objects) which can be used to make distributed decisions with other AIBOs. It takes care of the reasoning and of the transmission of data between the AIBOs. The only thing that other modules have to do is:

1) Suggest an action to the AiboReasoning module.
2) Wait for a 'go' or 'no go' and act accordingly.

The modules that make up AiboCommOpenR are:

- **AiboReasoning** - this is the module that does the actual reasoning, with the help of an expert system.
- **AiboCommIN** - listens for incoming data from other AIBOs.
- **AiboCommOUT** - this module is used by AiboReasoning to broadcast data to other AIBOs.
- **AiboDummyActionGenerator** - used as a test module in replacement of real life modules that are used on AIBO (e.g. from the soccer game). The current implementation starts the whole sequence of events when the back-sensor of AIBO is touched, and blinks its head-LEDs when a 'go' is received.
- **PulseGenerator** - is used as a timer for AiboCommOUT when it is broadcasting data and waiting for feedback.

An overview of the modules and how they communicate is presented in figure 9. In this figure, communication in point 1, 2, 6, 7, 9 and 10 is done using the 'observer/subject' structure described earlier, where 3 and 8 use normal UDP communication.

As shown in figure 9, the UDP communication in AiboCommOpenR is done in both the `AiboCommIN` and `AiboCommOUT` module. UDP is used here because it supports broadcasting of data, which is an important aspect of

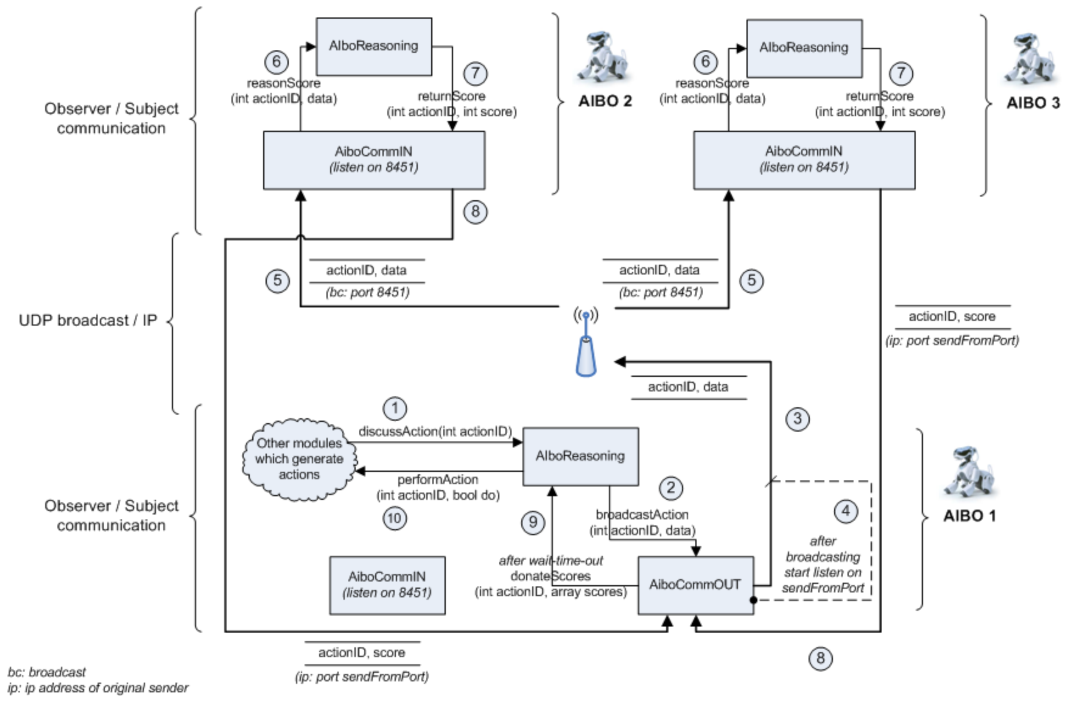

Observer / Subject communication
AIboReasoning
6
reasonScore (int actionID, data)
7
returnScore (int actionID, int score)
AiboCommIN (listen on 8451)
AIBO 2
AIBO 3
8
UDP broadcast / IP
5
actionID, data
(bc: port 8451)
actionID, score
(ip: port sendFromPort)
3
1
discussAction(int actionID)
Other modules which generate actions
performAction (int actionID, bool do)
10
2
broadcastAction (int actionID, data)
9
after wait-time-out donateScores (int actionID, array scores)
4
after broadcasting start listen on sendFromPort
AiboCommOUT
AIBO 1
bc: broadcast
ip: ip address of original sender

Fig. 9. Module overview of AiboCommOpenR

AiboCommOpenR. A transport mechanism for sending and receiving messages had to be designed for that communication. The protocol works on level 6 and 7 (presentation and application layer) of the OSI model, and because of that it needs to take care of the serializing and unserializing of the data that is transported between the AIBOs. A library is written which takes care of that, so that other modules can link to it and use AiboCommOpenR.

A predefined format is used for the serializing and unserializing of the data. AiboCommOpenR uses the following schema:

```
actionID::data-element1::data-element2
```

In `AiboCommOUT` the data elements are made up by data received from other modules. In case of the RoboCup soccer game, a data element could be the distance to the ball for example. In `AiboCommIN`, data elements are either the received data from `AiboCommOUT`, or a score (with or without confidence levels as described later in this section). The score that is meant here is the outcome of the reasoning engine as described in section III-D.

If more complex data has to be transmitted, the serializer can be easily extended to serialize complex structs as well. The current implementation of AiboCommOpenR only supports the struct in listing 1.

Listing 1. OTransportVectorData struct

```
#ifndef _DATATYPES_H_
#define _DATATYPES_H_

#include <vector>
#include <string>

using std::vector;
using std::string;

struct OTransportVectorData {

        int actionID;
        vector<string> data;

        void AddData(string dataToAdd) {
                data.push_back(dataToAdd);
        }
};

#endif //_DATATYPES_H_
```

C. Design Tradeoffs

For the design outlined above, certain tradeoffs had to be made. For instance, because of using UDP and wireless LAN, packets may get lost and therefore the system should not stall when not receiving any replies from other AIBOs. The system should not depend on the distribution of data to work. Because of this more or less unreliable transmission medium, we want the AIBOs to work stand-alone and not being controlled by a leader process or leader dog. Since data is distributed amongst all the AIBOs, this is not necessary anyway. All AIBOs can make decisions on their own, using their own expert system.

The sent and received data is merged into a summary of the team's knowledge about the current state. This state is referred to as the global world. Each dog has its own, local, copy of the global world. Looking at the soccer example for instance, this world contains information about the position of the ball, and position data of other AIBOs that was received earlier. That data might be out-dated and is updated when new data is received. For the sake of simplicity, no multi-hop is used and therefore AIBOs that aren't within each others reach won't receive each others data. However, this can be added later, in combination with history and confidence levels.

1) Using Multi-Hop: Multi-hop can be added to the system when the field that the AIBOs are operating in is larger then the reach of AIBO's wireless LAN, or when a lot of obstacles are around which can block the communication. For instance, if AIBO were to operate within a house with walls, using multi-hop would give an advantage as seen in figure 10. Using multi-hop in the soccer example would not give any advantage, since the playing field is small enough for the wireless signal to get to all AIBOs in the team, and there are no large obstacles on the field.

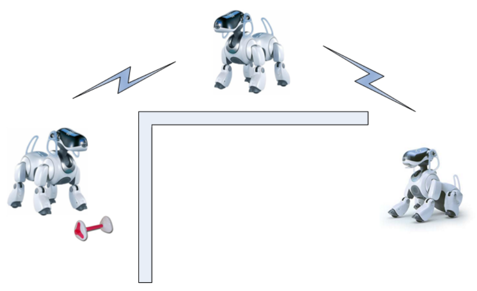


Fig. 10. Using multi-hop

2) Keeping History: History can be kept from the global world that was discussed above, to be able to make assumptions for the future. The history of the ball position (i.e. in the soccer game) for instance, can give the direction that the ball is heading. This also applies to the positions of the other AIBOs. Using their directions and the direction of the ball, AIBO can decide whether to attack and proceed in the direction of the ball, or stay put and defend.

When using AIBO as a watchdog, history of the received data can be used to determine whether to explore a certain room or head the other direction if other AIBOs have been there before. A condition for this to work, however, is that the history needs to be *shared* and *distributed*. This can result in consistency problems. We need not only keep one AIBO's data up-to-date, but the data shared between other AIBOs as well. This can give problems in case of failure of an AIBO, or when an AIBO leaves the network for battery charging purposes.

It may also pay off to save the results of the *reasoning process* for future use. Saving these results in combination with their outcome (i.e. whether the action was successful) can give an idea whether similar actions in the future will be successful as well. Broadcasts for help can be kept to a minimum this way.

The main problem with saving history is the fact that most data of the global world is outdated after a small period of time and therefore not usable anymore. Different applications also have different needs for saving history, and this history times out after different periods. For instance, ball positions in the soccer game are useless after a shorter period of time then information about rooms or walls in the watchdog example. Dealing with different applications therefore need different implementations of the history module. Therefore, history is not yet used in AiboCommOpenR.

3) Confidence Levels: Confidence levels can be used if it turns out that information from certain agents in the field is not reliable or accurate. They can be used either at the sending or at the receiving side, or even at both sides. If used at the sending side, they represent the confidence that AIBO has on the score that it sends back. A low confidence can be given if AIBO is unsure about some of his own data (e.g. his own position). This way the receiving AIBO can consider not using the received advise. A high confidence value, however, means that the receiving AIBO should use the advice.

If confidence levels are used at the receiving side, they represent the trust that the receiving AIBO has in the other AIBOs. In this situation, AIBO has an overview of all the agents and their corresponding trust values. This way AIBO can determine whether a received advise or score from a specific agent is reliable, and make a decision based on that. This trust value is calculated by the receiving AIBO itself, and is adjusted every time AIBO decides to use or not use a specific advise from an specific agent. Here a combination of both confidence levels can be used. Not using a score with a low confidence level can result in a higher trust value of the sending agent. The sending agent was not confident with his score and that results in a higher trust level for that agent. This is mathematically described in:

$$trust = clamp(0, 1, trust + \delta_p * (\gamma * trust * conf)) \quad (1)$$

where

$$\delta_p = \begin{cases} +1 & \text{if solution } s = \text{action } a; \\ -1 & \text{if solution } s \neq \text{action } a; \end{cases} \quad (2)$$

and $trust$ represents the trust value of an AIBO, $conf$ represents the confidence the sending AIBO has in his advise, γ is the trust learning rate, and $clamp(0, 1, trust, c)$ ensures that the value of c always lies in $(0, 1]$.[4][6]

This way an AIBO which proposes an incorrect advise with a high confidence value is penalized more heavily than one that proposes an incorrect advise but with a lower confidence value. Using confidence levels this way can result in a more reliable reasoning process, but also result in extra overhead, which is not desirable in this project because of the scarce resources of AIBO.

D. Designing CLIPS Rules

The CLIPS engine that is used for the reasoning part of AiboCommOpenR needs rules in order to reason. Different kinds of rules are needed for different kinds of applications. Since the focus of this paper is on the RoboCup soccer game, one example of such a rule for that application will be presented here.

1) Moving Towards the Ball: This example rule uses the AIBO's own distance to the ball and the distance from the other AIBO to the ball, and calculates a score with that data for the idea of moving to the ball that the other AIBO has.

The basic facts needed for this rule are:

- AIBOs own the distance to the ball
- the distance from the other AIBO to the ball (has been send along with the data)
- the default score to start with, for calculating the final score.

```
(defrule calculate_score_using_distance
    ``use distance''
  (not (calc_using_distance done))
  (distance self ball ?D_me_ball)
  (distance other ball ?D_other_ball)
  ?sc <- (score ?C_score)
=>
  (printout t ``Distance self-ball: '' ?D_me_ball)
  (printout t ``Distance other-ball: ''
   ?D_other_ball)
  (printout t ``Current score: '' ?C_score)
  (if (< ?D_me_ball ?D_other_ball)
  then
    (printout t ``I'm closer to the ball!
     decr score'')
    (retract ?sc)
    (assert (score (- ?C_score 1)))
  else
    (if (= ?D_me_ball ?D_other_ball)
    then
      (printout t ``Same distance to the ball,
       let him go! incr score'')
      (retract ?sc)
      (assert (score (+ ?C_score 1)))
    else
      (printout t ``He is closer to the ball!
       incr score'')
      (retract ?sc)
      (assert (score (+ ?C_score 1)))
    )
  )
  (assert (calc_using_distance done))
)
```

Testing this rule with a default score of 5, a distance of AIBO 1 to the ball of 10, and a distance to the ball of AIBO 2 of 20 results in:

```
CLIPS> (run)
Distance self-ball: 10
Distance other-ball: 20
Current score: 5
I'm closer to the ball! decr score
Final score is 4
CLIPS>
```

The score is correctly decreased by '1' since AIBO 1 (where CLIPS is running) is closer to the ball than AIBO 2 (the initiator). Therefore, AIBO 1 should be moving towards the ball instead of AIBO 2, and AIBO 1 should discourage AIBO 2 by returning a low score.

The rules that are currently designed for AiboCommOpenR are far from complete to be used in the RoboCup Soccer game. They are simplified rules, designed and implemented to be able to test AiboCommOpenR. If AiboCommOpenR is really to be used in the RoboCup soccer game, more complex rules are needed. Different rule sets can be written for different kinds of game play. Focusing on a strong defense for example, needs an other rule set then focusing on attacking. While the rules currently designed don't implement that yet, they do already take into account the different roles that each player can have in the team.

IV. Conclusion

This paper has provided an outline on the process of designing and developing a system of collaborative AIBO-robots. Test results of the current proof of concept implementation show that AiboCommOpenR is capable of communicating and sharing knowledge, and can use CLIPS on the AIBO for reasoning, and is thereby a complete framework for collaborative AIBO-agents.

It turned out that a complete reasoning-cycle took around 8300ms, which is not acceptable in most applications. However, disabling all the debug information (using the compiler-flag `-DOPENR_DEBUG` and `-DDODEBUG`) narrows this down to below 2 seconds. It seemed that writing the debug information to the memorystick, and writing the debug information to the console via wifi really slowed things down. Things can be speed up even more by lowering the wait timeout in `AiboCommOUT`. This way, AiboCommOpenR seems fast enough for the robots in the RoboCup game, and any of the other applications.

For the design, a few tradeoffs had to be made; whether the prototype should keep history, use confidence levels and/or use multi-hop. All off these were eventually not implemented, because of resource limitations, and/or were unnessecary for the RoboCup soccer game.

However, when things get more optimized, future implementations of AiboCommOpenR could include some of these aspects. It could include history, from either the outcome of the reasoning process or from the received data, and could use confidence levels. Using history, the AIBO can leave the event-driven path that it uses now, and act more pro-actively. With history, AIBO can make assumptions for the future, calculating the direction that the ball is heading for example, and keep broadcasts to a minimum when using the outcome of previous broadcasts for the current situation.

Other enhancements of AiboCommOpenR could include sharing AIBOs plans, or actively broadcasting information about the environment. For the soccer game example, this could include information on the ball position. This way all players in the field have knowledge over the ball position, even when their view is blocked, leading to a more active game by not having all the AIBOs scanning the environment. By sharing AIBO's plans in the RoboCup league, multiple AIBOs can cooperate and even play a one-two or use other passing techniques. Special defense and attack mechanism can be examined, and special formations can be tried, bringing the game to a higher level, using collaboration.

REFERENCES

[1] K. P. Sycara, "Multi-agent systems," School of Computer Science at Carnegie Mellon University," Project report for American Association for Artificial Intelligence, Summer 1998.
[2] Stuart J. Russel, Peter Norvig, *Artificial Intelligence: A Modern Approach*, 2nd ed. Prentice Hall, 2003. [Online]. Available: http://aima.cs.berkeley.edu/
[3] Dr. Michael Schroeder, "Software Agents - Introduction," http://agents.soi.city.ac.uk/introslides/tsld001.htm, Based on: H. Nwana, Software Agents: An Overview, Knowledge Engineering Review, 1996, 11(3):205-244.
[4] E. Daman BSc., "Collaborative robot agents," Research Assignment, Delft University of Technology, 2005.
[5] James Allen, Nate Blaylock, George Ferguson, "A problem solving model for collaborative agents," in *AAMAS '02: Proceedings of the first international joint conference on Autonomous agents and multiagent systems*. ACM Press, 2002, pp. 774–781. [Online]. Available: http://doi.acm.org/10.1145/544862.544923
[6] Yezdi Lashkari, Max Metral, Pattie Maes, "Collaborative Interface Agents," http://agents.www.media.mit.edu/groups/agents/publications/aaai-ymp/aaai.html.
[7] M.N. Huhns, M.P. Singh, "Distributed Artificial Intelligence for Information Systems," *CKBS-94 Tutorial*, June 1994.
[8] SONY Corporation, "OPEN_R_SDK-docE-1.1.5-r1," http://openr.aibo.com.
[9] Nicholas Roy, Gregory Dudek, *Collaborative Robot Exploration and Rendezvous*. Kluwer Academic Publishers, 2002. [Online]. Available: http://web.mit.edu/nickroy/www/papers/ARJournal.pdf
[10] Michael Wooldridge, Nick Jennings, "Intelligent Agents: Theory and Practice," *Knowledge Engineering Review*, vol. 10, no. 2, June 1995. [Online]. Available: http://www.csc.liv.ac.uk/~mjw/pubs/ker95/ker95.html
[11] M.R. Enesereth, S.P. Ketchpel, "Software Agents," *Communications of the ACMg*, vol. 37, no. 7, pp. 48–537, March 1994.
[12] Michael Weiss, "A gentle introduction to agents and their applications," http://www.magma.ca/~mrw/agents/.
[13] M.P. Papazoglou, S.C. Laufman, T.K. Sellis, "An organizational framework for cooperating intelligent information systems," *Journal of Intelligent and Cooperative Information Systems*, vol. 1, no. 1, pp. 169–202, 1992.
[14] Jean-Jack Riethoven, "Agents: the 'soft' kind," http://industry.ebi.ac.uk/~pow/Work/presentations/Agents/, March 2000.
[15] Tucker Balch, Lynne E. Parker, *Robot Teams: from diversity to polymorphism*. A.K. Peters, Ltd., 2002.
[16] "URBI," http://uei.ensta.fr/baillie/eng/urbi.html.
[17] "Tekkotsu Development Framework for AIBO Robots," http://www-2.cs.cmu.edu/~tekkotsu/.
[18] SONY Corporation, "[AIBO SDE] official web site," http://www.sony.net/openr/.
[19] R.E. Douglas, Jr., "Sneakers: A concurrent engineering demonstration system," Master's thesis, Worcester Polytechnic Institute, 1998.
[20] K. V. Laerhoven, "Comparison of the clips and jess expert system shells," Computing Department at Lancaster University," Project report for Industrial Applications of AI, June 1999.
[21] Dr. Franz J. Kurfess, "CPE/CSC 481: Knowledge-Based Systems," http://www.csc.calpoly.edu/~fkurfess/Courses/CSC-481/W03/Slides/2-CLIPS.ppt.
[22] PC AI Magazine, "PC AI - Expert Systems," http://www.pcai.com/web/ai_info/expert_systems.html.
[23] Anonymous, "Communications protocol," http://en.wikipedia.org/wiki/Communications_protocol.
[24] Radia Perlman, *Interconnections: Bridges, Routers, Switches, and Internetworking Protocols*. Addison-Wesley, 1999.
[25] Gerard J. Holzmann, *Design and Validation of Computer Protocols*. Prentice Hall, 1991.
[26] Novell, Inc., "Novell's Networking Primer," http://www.novell.com/info/primer/prim05.html.
[27] e. a. Jakob Ahln, Jonas Eriksson, "Gifr main report," Department of Information Technology, Uppsala University," An undergraduate project in the course Project DV, fall 2003.
[28] e. a. Pedro Cuesta-Morales, Zahia Guessoum, "Agent technologies at work," http://upgrade-cepis.org/issues/2004/4/up5-4Presentation.pdf, August 2004.
[29] Fabio Bellifemine, Agostino Poggi, Giovanni Rimassa, "JADE - A FIPA-compliant agent framework," http://sharon.cselt.it/projects/jade/papers/PAAM.pdf.
[30] "JADE - Java Agent DEvelopment Framework," http://jade.tilab.com/.
[31] Jeffrey S. Rosenschein, "An Introduction to MultiAgent Systems," http://www.csc.liv.ac.uk/~mjw/pubs/imas/distrib/powerpoint-slides/.
[32] Anonymous, "Multi-agent system," http://en.wikipedia.org/wiki/Multi-agent_system.
[33] Michael Wooldridge, *An Introduction to Multi-agent Systems*, 2nd ed. John Wiley & Sons Ltd, 2002.

©2008 Delft University of Technology