A CORBA-Based Distributed Software Architecture for Control of Service Robots

Steffen Knoop, Stefan Vacek, Raoul Zöllner, Christian Au, Rüdiger Dillmann
Institute for Computer Design and Fault Tolerance (IRF)
Universität Karlsruhe (TH)
Germany
Email: {knoop,vacek,zoellner,au,dillmann}@ira.uka.de

***Abstract*— This paper presents the distributed robot control software architecture developed for the autonomous service robot Albert2. The development of this architecture is focused on two major issues: Modularity and the integration of learning aspects. Each module within the architecture is presented, as well as the underlying event-based communication framework. An approach for integration of learning capabilities is proposed.**

I. Introduction

The architecture of a robotic system strongly influences the functionality and usability of such a system. In the last decades, many architectures have been presented which reflect the different requirements a robotic system has to cope with. Many requirements have to be considered during the design of a robot architecture, e.g. reactivity, which means the ability to react spontaneously to unexpected events, or the need for an adequate type of control flow. Other requirements are robustness, programmability, extensibility and adaptability amongst many more.

Extensibility and adaptability become very important if the robot has to act in a dynamic and partially unknown environment. This applies only in a very limited way to industrial robots, in contrast service robots are faced permanently with unknown or partially known situations. Due to the fact that service robots are not restricted to a well-defined working area but work together with humans in a human-centered environment, they must be prepared to cope with unknown, i.e. new, situations as well as to acquire and integrate new knowledge.

Extensibility can be reached through learning of task knowledge. Learning can be done offline or online, respectively, and it is desirable to have the possibility to add newly acquired knowledge to the robot's existent knowledge. Adaptability is the robot's ability to adjust itself to dynamic or even new environments. Furthermore it should be able to use its task knowledge if applicable even if the knowledge was learned in a different environment.

In this paper we concentrate on integrating into the architecture the robot's ability to learn task knowledge. On the one hand, this knowledge will be acquired offline through single user demonstrations, and on the other hand, the system should be able to adapt and to extend its knowledge at runtime. It is clear that these abilities have to be taken into account while designing a robot's architecture.

Section II gives a short overview of existing approaches, section III introduces our robot system Albert2, and section IV explains the learning approach "Programming by Demonstration". The proposed software architecture is presented in section V. First results are shown in section VI.

II. State of the Art

As has been stated by many research groups, e.g. [1], architectures form the fundament of robotic systems. Typically the existing approaches can be divided mainly into three different classes:

Hierarchical architectures are based on a top-down approach. Communication and controlflow is only done vertically between the different layers. Higher levels delegate subgoals to lower levels to achieve superordinate goals. An advantage of this approach is that planning is straightforward because there is a superior view on the system. A drawback of these systems is their insufficient reactivity to dynamical environments. Examples for hierarchical systems can be found in [2] and [3].

Behavioral systems (e.g. [4], see [5] for an exhaustive discussion) consist of several modules each representing a specific behavior. These behaviors are not grouped in a hierarchical manner but are running concurrently. These systems are robust because they don't rely on the existence of specific functional units compared to hierarchical systems. Because of their modular style single behaviors can easily be added. The lack of a high-level control unit complicates the planning to achieve certain goals. Moreover safety considerations are not easy to integrate.

Hybrid architectures try to combine the well defined control flow of hierarchical systems with the advantages of behavior based systems which are highly reactive. These approaches have become very famous recently and there exists a vast number of architectures with different foci (e.g. [6], [7], [8]) and [9].

Another aspect which must be considered during the design of a robot architecture is the learning ability of the system. Zhang and Knoll [3] use a hierarchical approach to learn operation sequences of two arm manipulations. Bonasso et al. (cf. [10], [11]) use an architecture organized into three layers: skills, sequencing and planning. Learning is possible in each layer and through different layers. Other

research work concentrates on learning a specific behavior [12] or mappings of sensor data [13].

III. Service Robot ALBERT2

The architecture which is presented in this paper has been realized on the service robot Albert2. Up to now it is mainly used in a kitchen environment and the focus of task learning and execution is laid on household environments.

Figure 1 shows the service robot Albert2 used for experiments in a household environment. For manipulation, it is equipped with 7 DoF arm and a three finger hand. A mobile platform serves for navigation. Built in sensors are a laser range sensor, a color stereo camera mounted on a pan tilt unit, and a force torque sensor mounted on the arm. For user interaction, the speech recognition system Janus [14] is used in combination with an external microphone, and loudspeakers on the robot provide speech output. Additionally, a touchscreen is attached to the front of the robot's upper body. It is used for displaying the robot's current internal state (e.g. waiting, working, idle) and for user interaction e.g by prompting questions on how to proceed (e.g. showing images of all graspable objects) to the user.

Fig. 1. Service robot Albert2

IV. Programming by Demonstration

In this section we shortly describe our approach for teaching a robot new task knowledge and how this knowledge is represented. A complete description of the learning mechanisms can be found e.g. in [15]. The approach is called *Programming by Demonstration* since the task is simply demonstrated by a human user in order to offer an easy to use interface for the unexperienced user. The focus lies on teaching higher level tasks since we assume that basic skills like specific grasps or basic motor control for movements of the arm already exist on the robotic system. We rely on one-shot-learning because users should not be forced to demonstrate the same task multiple times. The approach is basically composed of the following phases:

- Demonstration of the task
- Perception, data preprocessing and fusion
- Segmentation of the acquired data
- Generalization of the segmented data
- Simulation of the learned task and refinement of the task knowledge through user interaction
- Transfer of the newly acquired task knowledge onto the robotic system

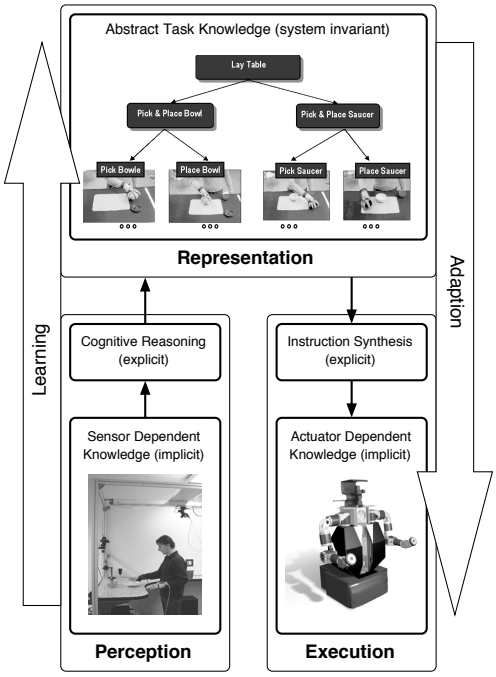

Fig. 2. Transfer of task knowledge of a human demonstration to a robotic target system.

The teaching starts with the user's demonstration in a specially designed training center where the performed task is perceived through different sensors like camera systems and data gloves. This center is neccessary because todays robot's sensors do not provide enough information with sufficiant precision. During the demonstration sensor data is merged with respect to the different sensor models. In the next stage the data is segmented depending on recognized grasps and movements. After that the data is analyzed and elementary operators which correspond to basic skills of the robot are identified. For generalization these basic operators are grouped hierarchically into so called macro-operators.

Finally the knowledge has to be transferred to the robotic system. Therefore we need a representation of the task knowledge for the robot which needs to be reflected by the architecture. Additionally a mapping of the macro-operators to the robot's task knowledge representations needs to be defined. The robot's representation must be extendable to be able to add new task knowledge and be adaptable in a way such that the existing knowledge could be refined either through reinforcement by the robot itself or by user interaction.

Figure 2 shows the learning process from user demonstration to generalized macro-operators on the left and the transfer of the task knowledge to the robot system on the right side. In the *Perception* and *Execution* phases

the type of background knowlegde which is used for data processing is indicated, e.g. the implicit sensor models used during the perception of the user demonstration, which enables data fusion of different sensor sources. In contrary, macro-operators are abstract and thus do not rely on this background knowledge.

V. Software Architecture

A software architecture that is able to control a mobile robot basically has to cover three aspects to comprise all capabilities of the system: Control of the hardware, representation of the environment and integration of knowledge about skill and task execution. Including the demand for a lifelong learning system, these requirements have to be fulfilled in a way that the system remains extensible.

An expedient approach to design such an architecture is therefore to identify the functional components as hardware abstraction, environment model and skill and task execution. Our proposed software architecture consists consequently of the following components (see figure 3):

The *hardware agents* encapsulate all hardware specific functions. There is a hardware agent for each hardware in the robot system, e.g. one for the robot arm, one for the hand and one for the voice. Skill and task knowledge is represented by the *flexible programs*. Example skills are ”drive to position” and ”open door”, tasks can be ”transport an object” or ”put object on the table”. All information about the environment is stored in an *environment model*. This can hold all kinds of data: Coordinates, objects, relations, features, images or sounds.

Evidently, there needs to be a way of communication for these components. The proposed communication bus seen in figure 3 allows communication between all components, but nevertheless standardizes and restricts data exchange to a defined set of data structures. It is event-based and is able to incorporate internal as well as external (user triggered) events.

This approach is very much inspired by the way an operating system works. The components are in detail:

- **The communication infrastructure** consists of a notification distribution instance, where clients can subscribe for certain notification types. Notifications may be delivered by internal or external sources.
- **Hardware agents** (resources) represent real or virtual sensors and/or actuators. There is an agent for *laser scanner* and for *cameras*, but there may be also an agent for *detection of humans* which incorporates laser scanner and vision information. Hardware agents are also referred to as *resources*.
- **The agent manager** administrates all hardware agents and provides the resource management. Each notification that is passed to an agent is filtered by the agent manager. Thus, unauthorized commands (from instances which have not locked the called resource) to agents can be intercepted.
- **Flexible programs** contain the skill and task knowledge. These flexible programs (*FPs*) are the core of the proposed robot control architecture. Learning, within our context, means creation, extension and adaption of flexible programs.
- **The flexible program manager** administrates the flexible programs. All notifications addressed to flexible programs are filtered and delivered by the flexible program manager. It also holds a list of all currently existing FPs, including their type.
- **Domain controlling and supervision** as well as FP instantiation and priority control is done by the flexible program supervisor. Depending on the current context, FPs are created, prioritized or deleted. In later development, learning capabilities will be extended to this component.
- **The environment model** holds environmental data as well as the robot’s internal state. It is implemented as a blackboard. An intelligent xml data base, which is being developed at our research group, will soon be integrated and used for storage of task, skill and object feature knowledge.

The implementation of the proposed software architecture (see figure 3) consists of a set of CORBA object types, which communicate via a communication instance using a publish-subscribe mechanism (see also [16]).

Each public object’s interface is described in CORBA’s Interface Definition Language, which enables the use of different programming languages within the same framework. We use C++ and have base classes implemented for each object type, which encapsulate all communicational and infrastructural aspects. This simplifies usage and implementation of new elements, as users do not have to know any details about communication and functionality of other objects.

Instantiating each component as a separate CORBA object allows object distribution over several computers and different operating systems. It also decouples operation of caller and callee, which is essential in complex systems. By running different system parts on different machines, it is on the one hand possible to split up required computation power, and on the other hand it enables different users to use the same infrastructure.

As most of the communicated information is very high-level symbolic information, real-time constraints are not defined primarily by data transmission, but by hardware restrictions; nevertheless, the use of distributed objects holds the risk of data loss or speed reduction.

Some of these architecture components will now be described in detail.

A. Communication layer

The basis for every distributed architecture is formed by the communication layer. The proposed architecture uses a special message format (referred to as *notifications*) as a basic information container for communication.

Notifications are small data blocks, wherein most of the information is coded as plain text. This is important for debugging and readability.

Notifications are always of one of three types:

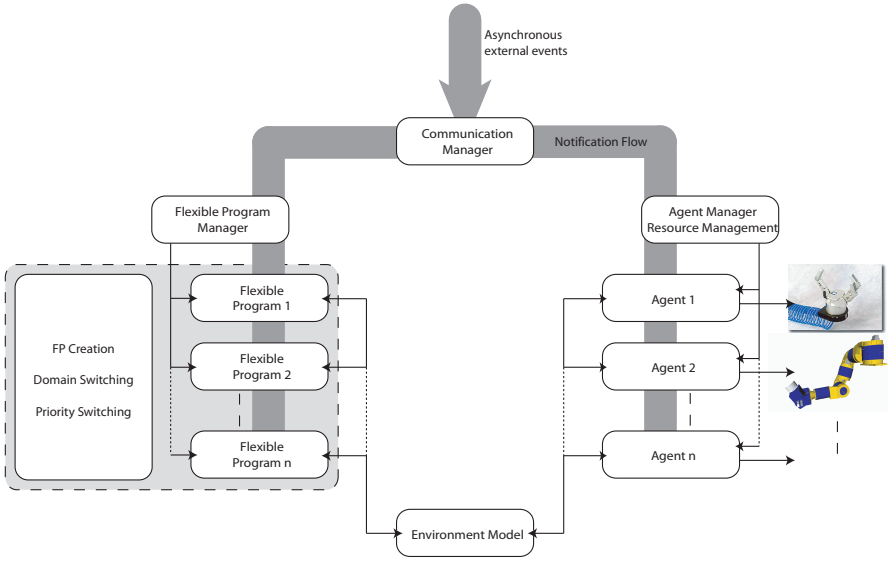

Fig. 3. CORBA-based software architecture with flexible programs, hardware agents and communication bus

- **Events** are always delivered to flexible programs. Events are generated either from hardware agents or from external sources like speech or gesture recognition. Events are data directed from sensors/actors to the control level.
- **Actions** are delivered to hardware agents. They are mostly generated by flexible programs. Actions correspond to data flow from the symbolic control level to the sensor/actor level.
- **Requests** are demands to the resource management and therefore delivered to the agent manager. Requests can be to lock, free, or try-lock resources.

TABLE I
NOTIFICATION DEFINITION

Header	
Type	Event, Action, Request
Time	Time and date of creation
Sender	Unique ID of sender instance
Receiver(s)	Unique IDs of receiver(s)
Body	
Name	Notification specification
key value list	Optional contents

A complete notification consists of a header and a body (see table I), the notification body again holds a name and a set of key value pairs. These have to be set by the sender and transmit the information. As every slot within a notification is defined in plain text, debugging and readability are simplified using a viewer and tools for manual creation and sending of notifications. Each notification name and the associated key value pairs have to be defined in advance. Thus, all inter-object communication is standardized and can be checked for consistency and conformity with the specification.

To control the data flow of notifications and to avoid storage of unattended messages, the timestamp in the notification header controls notification deletion from the delivery queues.

Notification distribution is performed by the communication manager using a publish-subscribe mechanism. Undelivered messages are not stored within the communication manager, but discarded directly. In contrast, the agent and flexible program manager possess a notification queue, wherein messages are stored until they are either delivered or expired.

B. Resource management

Real and virtual hardware agents are also referred to as resources. These resources have to be locked and unlocked before usage from any flexible program to avoid collisions. This resource management is done by the agent manager, which also administrates the hardware agents. Agent administration comprises keeping track of the agent's state (*running*, *sleeping*, *stopped*), of current locks, of safety aspects (stop/resume all agents in case of emergency stop) and also notification surveillance and distribution among hardware agents.

Agents can only be locked by one flexible program at a time, but these usage authorizations can be inherited.

C. Hardware agents

Each hardware agent is identified by a unique ID and a unique agent type. Depending on the agent type, an agent can accept different notifications: A vision agent is able e.g. to search objects, while an robot arm agent can move or switch move mode.

The capabilities of an agent are context independent and defined by the capabilities of the underlying (real or virtual)

hardware. The hardware agent for the mobile platform e.g. accepts actions for geometric and topological navigation: *Rotate relative/absolute*, *drive relative/absolute*, *set position*, *stop driving* as well as *drive to node X*.

D. Interruption and exception handling

There are two cases, where running operations have to be canceled immediately:

- The goal has changed while an operation was running. This goal change can be caused by a human or by other external events. The goal change has to be carried out as ”smooth” as possible.
- An emergency situation occurs. This can either be an internal one (like collision detection, or some other conflict) or invoked by the user (e.g. saying ”stop” or pushing a soft stop button). It is very important that these exceptions trigger a stop of all hardware components immediately and that the system remains passive until an explicit resume command occurs.

While the latter is realized through direct stop calls (which are not handled by the standard communication and queuing software), to achieve the former, a continuous goal adjustment has to be performed within the hardware agents.

E. Flexible programs

Flexible programs always have the following properties:

- A unique ID, used to identify the FP within the whole system.
- A state (inactive/active, when active: FSM-like state description).
- A list of notifications that are accepted at the current state.
- A priority which can be used for the decision which FP gets a notification that is accepted by more than one FP.
- A list of currently owned resources.

These properties are used, set or read by the flexible program manager and the domain controller and are needed for communication.

Flexible programs encode the task knowledge. They generate agent commands from background knowledge, environmental data, and robot internal states. FPs also have to handle errors.

The flexible program description is independent from the robot’s kinematics, as all hardware specific algorithms are encapsulated in the hardware agent objects. Of course, to keep two robots exchangeable, they must possess similar hardware components.

F. Flexible program specification

Within our concept, the flexible programs hold the task knowledge that is included in the system. This knowledge is represented as the number, type, parameterization and order of the basic skills used. We propose to use a meta programming language. This task description language has to fulfill several requirements:

- It has to be powerful enough to describe all possible action sequences as well as dependencies, decisions and exceptions.
- On the other hand, this language should make as many restrictions as possible to ensure ease of use and debugging capabilities as well as to avoid ambiguities.
- It should be easily convertible to a standard data description format like xml. This enables use of standard tools for saving, searching, comparing and merging different flexible programs.
- To guarantee extendibility at runtime, the language must not need a compilation process. A program should be directly executable.

There are languages that fulfill some of these requirements (e.g. *ESL*, see [17], or *TDL*, see [18]), but in particular the program extendibility has not been an explicit demand.

G. Task knowledge representation

A flexible program holds task knowledge, which is encoded in its specific structure and parameters. The implementation of an FP which provides the interface and functionality described in section V-E is proposed as follows:

A flexible program is composed of functional blocks. These functional blocks define functionality, pre- and post-conditions in the same way FPs do; in fact, often FPs are used as functional blocks within other FPs.

One functional block consists of an input parameter list (coordinates, object names, persons, area of interest, etc), different result states (several success and failure states) and their output parameters (positions, objects, persons, etc), and the immanent functionality. This can consist of one or more calls to hardware agents or other flexible programs, including requests to the environment model to get or set required or perceived data and results. Such a functional block comprises manipulation and/or perception, the environment (including the robot itself) is always in the loop. It also contains information about interruptibility. If the current block is interruptible, there are additional suspend and resume methods.

Inside a flexible program, these blocks are then wired according to the task specification and depending on the respective results. As this sequence is a high level action chain, it is very easy to understand and debug.

It is important to note that there must not exist concurrent, asynchronous processes within one FP. This assumption has to be made to avoid collisions in resource management. This case has to be solved with different flexible programs.

H. Generation and extension of flexible programs

As flexible programs are defined on a high abstraction level using functional blocks, generation and extension is possible offline and online. Basically, there are three ways to integrate new knowledge into the system:

1) Manual FP definition: Flexible programs can be defined by a programmer. This can be done either using a GUI to build and connect functional blocks or by coding by hand.

2) FP derivation from user demonstrations: Flexible programs can be generated from macro operators (see section IV). Macro operators are set up by observation of a human demonstrating a task. This observation can either be done offline using extra hardware and software or online by the robot itself. Currently, a special demonstration environment is used with special sensors and software.

This transformation process will presumably be carried out semiautomatic; the operator will still need to correct and modify robot programs that are generated automatically.

3) FP extension by user interaction at runtime: Robot programs that are not complete, i.e. not every possible outcome situation is covered by the FP, can be extended at runtime. If such a situation occurs, the system can ask the user for a solution. Then, an empty functional block is constructed and filled by joining information given by the user with background and context knowledge. If this extension leads to a satisfying result, the FP is saved and used at the next time.

If e.g. the robot fails to move to the next room because the door is closed (and closed doors are not modeled as obstacles yet), the user can tell it to open the door and try again. This knowledge is then saved and reused whenever this situation occurs.

VI. Results

The architecture has been successfully implemented and tested on the service robot "Albert2". The robot is able to execute tasks like fetch-and-carry or pick-and-place operations successfully.

The required skills and time needed for implementation of additional functionality was drastically reduced, because the chosen architecture and knowledge representation supported the programmer by encapsulating most low level and infrastructural issues related to communication, timing, resource management, etc.

VII. Conclusions and future work

In this paper, we presented an approach for a service robot architecture. The way task knowledge is represented within the architecture allows for permanent extension of the knowledge base. Knowledge is perceived through user demonstrations, and can be extended at runtime by the robot itself or by user interaction. Taking these options into account, a suitable task knowledge representation has been proposed, where task knowledge is described by *flexible programs*.

The proposed learning capabilities will now be implemented and integrated in the presented framework. The architecture will be extended by a task planner, which will be incorporated into the learning process.

Acknowledgment

This work has been partially supported by the german collaborative research center "SFB Humanoid Robots" granted by Deutsche Forschungsgemeinschaft (see `http://www.sfb588.uni-karlsruhe.de`) and by the European Union via the Integrated project "COGNIRON (Cognitive Robot Companion)" funded under the Sixth Framework Programme (see `http://www.cogniron.org`).

References

[1] Ève Coste-Manière and R. Simmons, "Architecture, the backbone of robotic systems," in *Proc. 2000 IEEE International Conference on Robotics and Automation*, San Francisco, CA, April 2000.

[2] J. Albus, R. Lumia, and H. McCain, "Hierarchical control of intelligent machines applied to space station telerobots," in *Transactions on Aerospace and Electronic Systems*, vol. 24, September 1988, pp. 535–541.

[3] A. K. J. Zhang, "Control architecture and experiment of a situated robot system for interactive assembly," in *Proc. 2002 IEEE International Conference on Robotics and Automation*, Washington, D.C., May 2002.

[4] S. A. Blum, "From a corba-based software framework to a component-based system architecture for controlling a mobile robot," in *International Conference of Computer Vision Systems (ICVS 2003)*, Graz, Austria, April 2003, pp. 333–344.

[5] R. C. Arkin, *Bahvior-Based Robotics*. Cambridge, Massachusetts: The MIT Press, 1998.

[6] R. Alami, R. Chatila, S. Fleury, M. Herrb, F. Ingrand, M. Khatib, B. Morisset, P. Moutarlier, and T. Siméon, "Around the lab in 40 days ..." in *Proc. 2000 IEEE International Conference on Robotics and Automation*, San Francisco, CA, April 2000.

[7] K. H. Low, W. K. Keow, and M. H. A. Jr., "A hybrid mobile robot architecture with integrated planning and control," in *Proc. of the First International Joint Congerence on Autonomous Agents and Multi-Agent Systems*, Bologna, Italy, July 2002, pp. 219–226.

[8] M. N. Nicolescu and M. J. Mataric, "A hierarchical architecture for behavior-based robots," in *Proc. of the First International Joint Congerence on Autonomous Agents and Multi-Agent Systems*, Bologna, Italy, July 15-19 2002.

[9] P. Elinas, J. Hoey, D. Lahey, J. D. Montgomery, D. Murray, S. Se, and J. J. Little, "Waiting with josé, a vision-based mobile robot," in *Proc. 2002 IEEE International Conference on Robotics and Automation*, Washington, D.C., May 2002.

[10] R. P. Bonasso and D. Kortenkamp, "An intelligent agent architecture in which to pursue robot learning."

[11] R. P. Bonasso, J. Firby, E. Gat, D. Kortenkamp, D. P. Miller, and M. G. Slack, "Experiences with an architecture for intelligent, reactive agents," *Journal of Experimental & Theoretical Artificial Intelligence*, vol. 9, no. 2/3, pp. 237–256, April 1997.

[12] W. Paquier and R. Chatilla, "An architecture for robot learning," in *7th International Conference on Intelligent Autonomous Systems*, Marina del Rey, California, USA, March 2002.

[13] M. Sahami, J. Lilly, and B. Rollins, "An autonomous mobile robot architecture using belief networks and neural networks," in *working paper, Department of Computer Science, Standford University*, 1995.

[14] H. Soltau, F. Metze, C. Fügen, and A. Waibel, "A one-pass decoder based on polymorphic linguistic context assignment," in *IEEE Workshop on Automatic Speech Recognition and Understanding (ASRU '01)*, Madonna di Campiglio, Italy, 2001.

[15] M. Ehrenmann, R. Zöllner, O. Rogalla, S. Vacek, and R. Dillmann, "Observation in programming by demonstration: Training and execution environment," in *IEEE International Conference on Humanoid Robots*, Karlsruhe, Germany, Octobre 2003.

[16] R. Dillmann, "Interactive natural programming of robots: Introductory overview," in *DREH 2002*. Tsukuba, Ibaraki, Japan: Tsukuba Research Center, AIST, December 2002.

[17] R. P. Bonasso, J. Firby, E. Gat, D. Kortenkamp, D. P. Miller, and M. G. Slack, "Experiences with an architecture for intelligent, reactive agents," vol. 9, no. 2/3, Apr. 1997, pp. 237–256. [Online]. Available: citeseer.ist.psu.edu/bonasso97experiences.html

[18] R. Simmons and D. Apfelbaum, "A task description language for robot control," 1998. [Online]. Available: citeseer.ist.psu.edu/simmons98task.html