Proceedings of the 2004 IEEE
International Conference on Robotics & Automation
New Orleans, LA • April 2004

A Distributed Architecture for Executing Complex Tasks with Multiple Robots

Topi Mäenpää, Antti Tikanmäki, Jukka Riekki, and Juha Röning
Intelligent Systems Group and Infotech Oulu
Department of Electrical and Information Engineering
University of Oulu, Finland
Email: {topiolli, sunday, jpr, jjr}@ee.oulu.fi

Abstract— **This paper presents a software architecture for the network-transparent control of distributed robotic systems. The system consists of two main components: a generic and easily extensible CORBA-based interface to distributed services, and a high-level XML-based description language for specifying the behavior of the robots. The architecture makes it possible to create dynamically modifiable, extensible control software with ease. It is successfully utilized in implementing a coffee serving system in which the co-operation of two very different robots and two other distributed services are needed.**

I. Introduction

At the moment we are witnessing the break-through of robots helping people in various tasks. A number of service robots have been recently introduced, including an autonomous vacuum cleaner and a servant robot [1]. The mission of these robots is to serve people in their everyday lives and hence increase people's quality of life. In addition, a number of robots have been build for pure entertainment. However, to be really attractive the robots should provide much more functionality. They could perform a wider variety of tasks and more complex tasks as well. This can be achieved by increasing the number of robots. Through co-operation, the robots can perform tasks together, each specializing in its own expertise area. The individual robots can be relatively simple and the resulting robotic system is more robust and cheaper than one robot providing the same functionality.

The diversity of different types of robots makes it very important to design interfaces that are easily extensible and straightforwardly implementable. An abstraction of behaviors is important to allow the use of a control software with different robots without a need to rewrite code for each. The same also applies to other distributed services like cameras and other sensing instruments. With a universal interface and high-level action abstraction, it is possible to use one software to control a wide variety of different services.

Designing the behavior of a complex distributed system with many service robots is a non-trivial task. To facilitate the analysis of a control software, well-defined design principles are needed. For this purpose, finite state machines (FSM) are a good choice [2]. With FSMs, the control software can be designed on a high level, even with graphical user interface tools. Unlike pure visual languages (see e.g. [3]), FSMs are well suited for defining also low-level behavior. They allow painless modularization of the control software, and make it relatively easy to test just parts of the whole system. Finally, due to their general nature, FSMs can be used to implement many different kinds of control architectures. For example, the subsumption architecture of Brooks [4] can be implemeted with hierarchical FSMs. The FSM approach also has some shortcomings, some of which can be eliminated with the proposed approach.

In this paper, a dynamically modifiable FSM-based control architecture, logical sensors [5], and logical actuators [6] are put together. We present a universal interface to distributed services and a high-level language for describing the behavior of the co-operating robots. The interface, called *Property Service*, is based on CORBA, and the language on FSMs. Although there are quite a few software design tools capable of building executable code out of FSMs, we have chosen a different approach that allows the control software to evolve during execution. The behavior of a robot can be designed completely independent of the actual control software, which needs no recompilation or even restart. This way, the control software can be kept small, static, and simple.

In the proposed *Stemach* architecture, an XML/RDF description is all that is needed to build and execute an FSM. No compiled code is needed, and the state machines can be dynamically modified. Compared to other approaches, the architecture presented in this paper is simpler, yet having more expressive power. Due to the standardized description language and interfaces, and a modularization mechanism, the architecture is also very easily extensible.

II. Interface to Distributed Services

A major problem in controlling sofware-driven distributed services is in designing and maintaining interfaces. If a software needs to be capable of accessing, say, two different robots, a vision system, and a distributed message service, it may need to handle four very different interfaces. Whenever new features are added to the services, the interfaces need typically to be changed accordingly. If the same services need to be accessed with different programming languages, the interface must be changed for each. As a solution to these problems, a generic, simple and extensible interface called Property Service is proposed.

TABLE I

THE STANDARD SET OF READ-ONLY PROPERTIES EVERY PROPERTY SERVICE SHOULD IMPLEMENT

Property	Description
type	The type of the server. This may be any user-defined string. Possibilities range from ”robot” to ”thermostat”.
name	The name of the server. Helps client in distinguishing between multiple servers of the same type.
version	The version of the server software, e.g. ”1.0”.
properties	An XML-encoded list of known properties. This list should contain all standard properties and any other properties known to the server.

In a sense, the Property Service builds on top of the idea of logical sensors [5], [7]. Our approach does however consider not only sensors but also other distributed services, which may be considered logical actuators [6]. By logical actuators we mean devices that provide some functionality. Just like with logical sensors, it is irrelevant to the controlling application how this functionality is physically performed. For example, a mobile robot may be capable of moving an object from one place to another, but the same functionality may also be provided by a stationary manipulator. If an application needs to have the object moved, it may not care how the movement is actually performed.

Recently, Wang *et al.* presented a COM-based architecture for the fusion of logical sensors [8]. The approach does hovewer have some drawbacks, including platform dependency and the lack of network transparency. We are trying to overcome these by using CORBA as the component model.

Property Service is a general representation of a network-transparent server that is controlled using property name-value pairs. The server is accessed via CORBA calls, and the interface consists of just two methods: one for setting the value of a property and another for retrieving it. The interface was designed to be as simple as possible to facilitate easy exploitation. Thus, the approach is quite different from that chosen for example for the Mobility™ software [9] or the CORBA notification service [10].

Property names are presented as human-readable strings, and used to refer to the logical functions of a distributed service. Property values in turn can contain any application-specific data, ranging from simple sensor measurements to video frames. The system also makes it possible to register *listeners* to specific properties. The listeners are notified each time the value of a property changes. The IDL (Interface Definition Language) definitions for the Property Service and the listener are as follows:

```
module propertyservice {
  module corba {
    exception PropertyException {
      string message;
    };
    typedef sequence<octet> OctetSeq;
    interface PropertyService {
      void setProperty(in string propertyName,
                       in OctetSeq propertyValue)
        raises ( PropertyException );
      OctetSeq getProperty(in string propertyName)
        raises ( PropertyException );
    };
    interface PropertyServiceListener {
      void propertyChanged(in string propertyName,
                           in OctetSeq propertyValue);
    };
  };
};
```

The functions of the methods can be broadly summarized as follows. The `setProperty` method is used in configuring sensors and actuators, and in setting goals to actuators. The `getProperty` method in turn is used in obtaining sensor measurements and state information. The `propertyChanged` method is used in informing registered listener of changed property values.

Property names are organized in a hierachical structure in which components are separated with dots, just like with domain names. For example, `position.relative` and `position.absolute` are used to get or set the relative and absolute positions of a mobile robot.

In designing distributed services, performance issues must be considered. Therefore, it is better to deliver refined information rather than raw data. For example, instead of delivering captured image data as a property value, the images are processed locally and only measurement results are delivered. This decreases the required amount of data dramatically and increases the performance. The obvious drawback is that more computation hardware is needed in the distributed devices.

As a portable way of encoding complex property values, XML-based descriptions are used. While this choice imposes some overhead on the system, it also makes it very flexible. Different types of properties may be added at will, and there is never a need to redesign the interface. The XML descriptions can be converted to data structures in many different ways, depending on the programming language used. With C++, templates are used, and the conversion is fully transparent to the application programmer.

A “standard” set of read-only properties, shown in Table I, is defined to allow applications to make sure how to deal with a service, and to work with different versions of a service. Each specialization of the basic Property Service adds to these standard properties. In the following, an application fetches an XML-encoded list of all properties known to a Property Service:

```
props = getProperty("properties");
```

In this and all subsequent examples, object references are omitted for brevity, and because the calling syntax is dependent on programming language. In C++, for example, the statement above would most likely be something like `service.getProperty("properties")`, in which `service` is an instance of an CORBA remote object stub.

A connection to a Property Service can be established in two alternative ways. The object request broker (ORB) assigns a unique inter-operable object reference (IOR) string to each remote object. This can be passed to clients by any means, for example through a file in a shared file system. Another, more

convenient way is to use the CORBA naming service. Each server registers itself to the naming service, which passes their object references to clients.

Since the Property Service interface is based on CORBA, it is quite straightforwardly implementable in many programming languages. Currently, there are implementations for C, C++, Java, and PHP.

In Sections II-A–II-D, four different specializations of the Property Service interface are briefly described. All these were used together in the experiment presented in Section IV-C.

A. Mobile Robot Service

The mobile robot service provides a set of properties for controlling the movement and sensors of a robot. The top-level interface contains properties that are common to every mobile robot: movement (translation, rotation, and speed) and sensors measuring the internal state of a robot and various external features like distances to obstacles. Each specialization of the mobile robot service implements at least these functionalities. The services may also provide additional functionality specific to a certain robot only. Through the Property Service interface, different sets of properties can be easily provided, while the basic set is always available.

Mobile robot servers may also provide properties for time critical calculations and common mobile robot algorithms like collision avoidance using free space measurements in the local environment. Furthermore, some robots provide alternative ways for the execution of the basic commands. For example, the method of movement may be not only `direct drive` (the default) but also `virtual force field` (VFF) or `neural net`. Collision-free path planning using a virtual force field was used in the practical example discussed in Section IV. Currently, we have implementations of the mobile robot service for Pioneer 2, Super Scout 2, Nomadic XR4000, and Qutie. Qutie is a friendly-looking robot built on top of a Super Scout 2. It includes a 2-DOF head and a belly display for showing emotions. The services for Pioneer 2 and Super Scout 2 (or Qutie) can also be used as front-ends to simulators.

In the example below, two uses of the `setProperty` method are demonstrated. First, an actuator is configured by setting its method of movement to VFF. Then, a goal is set that makes the robot to try moving two meters forward and to the left. During the movement, the state of the robot or its sensors can be inspected as shown by the last row. The `getProperty` call returns the measurements of eight sonars. On a higher abstraction level, an application could as well request the distance to an obstacle at a certain direction. The three dimensional vector in the example represents the x and y coordinates of the robot, and the rotation angle with respect to the current position.

```
setProperty("moving_method","VFF");
setProperty("position.relative",
            "<vector size='3'>2 2 0</vector>");
sonars = getProperty("sonar.[0-7].measurement");
```

B. Message Service

As a convenient way of passing messages between different service controllers, a simple message-passing service was built on top of the Property Service. Although there are a number of different message passing services available, we selected to implement our own. The reason for this is two-fold. First, many of the alternatives, like the CORBA notification service [10], are very complex. In the application presented in this paper, only simple messaging is however needed. Second, we wanted to bring the message-passing system into the Property Service framework to make all services accessible through the same interface.

A client requests message delivery by setting the `register` property on the message service. As a parameter, it sends the server its unique ID (selected manually) that other clients use in indentifying the receiver of their messages. A message is sent to a registered client by setting the `message.ID` property where `ID` is the ID of the receiver. The message itself may contain any application-specific data. A typical sequence of operations in passing messages between two clients (A and B) is the following:

```
A: setProperty("register","A_ID");
B: setProperty("message.A_ID", "message");
A: msg = getProperty("message.A_ID");
```

In this particular case, A gets the message by calling the message service. Another, more convenient way is to register a property listener for incoming messages. This way, the message is passed to the client immediately after it has been received by the message service.

In principle, a single control software (FSM) could be used in controlling all the services in an application. The message service would be unnecessary because messages could be sent internally in memory. In practice it is however unlikely that everything is controlled by one application, and a way of passing messages between controllers is needed.

C. Vision Service

In all our mobile robots, vision is an important source of information. The vision services are implemented on top of the Video4Linux and Video4Linux 2 interfaces, which cover most of the frame grabbers currently available. Raw image data is obtained through the low-level interface, and the vision service extracts the needed features. The extracted features range from color and texture descriptors to detected lines and circles. In the application described in Section IV, a very specialized vision service was used in detecting a vacuum jug and coffee mugs. For example, an application can get the 2-D coordinates of the tip and lid of the vacuum jug in the manipulator's coordinate system with:

```
jug = getProperty("jug.manipulator");
```

A vision service should also be capable of sending raw image data for example for remote control purposes. The services may also provide the ability to control the orientation of the camera, a zoom lens, the aperture, etc.

D. Manipulator Service

A specialization of the Property Service interface has also been created for controlling a six-axis robotic manipulator. The manipulator service provides properties for getting and setting the position of the tip of the manipulator in world and joint coordinates. It also allows one to inspect the state of the manipulator, and to open and close a gripper operated with pressured air. The software controls the manipulator via an RS-232 interface.

The position of the manipulator is presented as a six-dimensional vector. In world coordinates, the elements of the vector contain the x, y, and z coordinates and three rotation angles. In "joint" coordinates, the elements denote the angles of each joint. In the example below, the position of the manipulator is obtained and set using the "position" property. It is also shown how the gripper is operated:

```
position = getProperty("position.world");
setProperty("position.world",position);
setProperty("gripper","close");
```

III. Specifying Robot Behavior

Finite state machines are a convenient way of designing the behaviour of a robotic system. State machines make it relatively easy to design complex behavior including nested actions and parallelism. The approach does however have some shortcomings, the most obvious of which is the finiteness of the number of states. Not all types of software can be described with a finite number of distinct, well defined states. Some researchers have presented hybrid models in which state machines are used on a low level whereas high-level tasks are defined in a different way. An example of such a system is the task description language of Košecká *et al.* [11]. We wanted to keep the design of our system consistent so that the same type of description can be used on each abstraction level. This approach does however limit the implementation techniques. Even when a control algorithm cannot be implemented with FSMs, it can still be initiated from an FSM.

The problem with most current state machine design tools is that they are not able to dynamically evaluate or modify the state machines. Instead, state machine descriptions are converted into some programming language, compiled and executed as static code (see e.g. [12]). From the computational performance point of view, this is not a bad solution, but it fixes the structure and parameters of a state machine at design time.

Well-defined ways of constructing hierarchical state machines already exist. Perhaps the most commonly accepted description is embedded in UML [13]. It provides a way of describing hierarchical states and their actions on a rather abstract level. The description syntax also supports parallelism in the form of "orthogonal regions" inside states. The concept of a state machine in UML is tightly bound into class descriptions, which is a logical consequence of the fact that UML is an object-oriented design language.

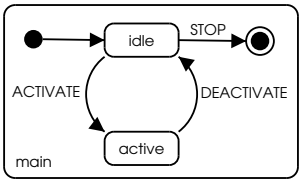

Fig. 1. A simple hierarchical state machine

A. Dynamical State Machines with Stemach

The Stemach state machine architecture we are proposing offers an alternative to the compiled code approach. Stemach uses a language based on XML/RDF to represent state machines that are dynamically evaluated at run-time. There is neither a need to convert the description to a programming language, nor does the description require compilation. We call the description syntax SMML (State Machine Markup Language). As a globally accepted standard, XML is very well suited for describing distributed systems, the design of which must be accomplished with many collaborating parties. Recently, Kim *et al.* proposed this kind of co-operation model for the development of interactive robots [14]. XML has been used in defining the behavior of mobile robots by Blank *et al.* [15]. Their approach, termed XRCL, is however strongly based on C++ and compiled code. No state machines are involved. Instead, the design of behaviors is based on fuzzy logic.

An XML document is all that is needed to construct and run a Stemach state machine. An executing state machine can be dynamically modified or replaced with another. Stemach also provides a true state hierarchy on execution level. All nested states are executed in a virtually parallel fashion. The UML concept of orthogonal regions is dismissed, and states are allowed to be active without restrictions.

Modularity is a key issue in specifying complex actions. The Stemach architecture allows one to design separate actions completely independent of each other. Through the hierarchical execution mechanism, it is easy to import complex actions that appear as single states on a high-level state machine. The components of a state machine can be stored in separate files, accessible in the local file system, via HTTP, or via FTP. Therefore, it is even possible to dynamically generate state machines on a web server. This allows robots to dynamically change parts of their behavior without the need to hard-code a gigantic control software that takes everything into account.

B. An Example

Figure 1 shows a very simple state machine consisting of a compound state called `main`. This is a convention when creating state machine descriptions. The main state works as a starting point other state machines can use when referring to external SMML descriptions. The main state in this particular machine contains two states in addition to the start and end states. Since there is no action associated with the transition from the start state to `idle`, `idle` can be directly defined as a start state. The functionality of the `active` state is actually

defined in another SMML document called `external.xml`, and this state machine treats it as a single state. An Uniform Resource Identifier (URI) is used to refer to the external state machine, in this case `external.xml#main`. Events coming from an external source drive the state transitions in the machine. The SMML description of this state machine is as follows:

```
<?xml version="1.0"?>

<!DOCTYPE smml [
<!ENTITY isgns
 "http://www.ee.oulu.fi/isg/stemach/1/schema#">
]>

<rdf:RDF xmlns:rdf=
         "http://www.w3.org/1999/02/22-rdf-syntax-ns#"
         xmlns:isg="&isgns;"
         xmlns="&isgns;">

<state rdf:ID="main">
  <substates>
    <rdf:Seq>
      <rdf:li rdf:resource="#idle"/>
      <rdf:li rdf:resource="#active"/>
      <rdf:li rdf:resource="#end"/>
    </rdf:Seq>
  </substates>
</state>

<state rdf:ID="idle">
  <type>start</type>
  <transitions>
    <rdf:Seq>
      <rdf:li><transition isg:event="ACTIVATE"
                          isg:target="#active"/>
      </rdf:li>
      <rdf:li><transition isg:event="STOP"
                          isg:target="#end"/>
      </rdf:li>
    </rdf:Seq>
  </transitions>
</state>

<state rdf:ID="active">
  <extend rdf:resource="external.xml#main"/>
  <transition isg:target="#idle"/>
</state>
<include isg:uri="external.xml"/>

<state rdf:ID="end" isg:type="end"/>
</rdf:RDF>
```

The ACTIVATE and STOP events may come from virtually any source. For example, a concurrently running state machine could fire these events based on messages received from a message service. They may also be signals from the operating system, or from a user interface.

As such, the state machine of Figure 1 performs no useful job as neither state activities nor any actions are specified. It just changes state according to the two events. In a realistic setting, the actions are used to perform calculations and to invoke remote procedure calls to distributed services.

C. Implementation

The SMML specification itself does not place any limits on the implementation technique of the software executing the state machines. State activities and actions associated with state transitions can be written in any programming language. The implementation we have been using is written with the PHP scripting language, although any other language could do as well. The only requirement is that the actions and activities in a state machine must be evaluated at run-time. Our PHP implementation has an *external execution* mode in which all actions and activities are passed to an external software. This allows one to use many different programming languages in defining the actions and activities of a state machine.

In the terms of XML/RDF, each entity (state or transition) in the state machine description is called a *resource*, each identified by a unique URI. Upon parsing the description, the software forms a set or *statements* about the resources. Each statement is a tuple of the form $< predicate, subject, object >$, in which $predicate$ can be treated as the name of a property, $subject$ as a reference to a resource, and $object$ as the value of the property. An example of such a statement could be $< action, \#transition1, exit >$, i.e. "The *action* associated with the transition identified by *#transition1* is *"exit"*. By altering the set of statements, it is possible to change the behavior of a state machine even during its execution. For example, to change the state transition action in the previous example, one would need to change just the $object$ of a single statement. It is also possible to add, delete and modify states and transtions.

IV. Serving Coffee with Robots

As a working example of a complex distributed system, a coffee-serving robot was implemented. The system consists of a "waitress", which can be any mobile robot, a manipulator that delivers coffee from a vacuum jug, a vision service for detecting the vacuum jug and coffee mugs, and a message service for passing messages between the state machines controlling the mobile robot and the manipulator. The overall architecture of the system is illustrated in Figure 2 (a). It is irrelevant to the control software where in the network the services are actually located. In our case, the Stemach control programs, the image analysis service, and the message service were all run in a single computer. The services controlling the mobile robot and the manipulator were dedicated one computer each. The network connection to the mobile robot was arranged through a wireless LAN.

A. Coffee Machine

Our high-tech coffee machine (Coffee-O-Bot) consists of a GMFanuc S-10 six-axis industrial manipulator, and a vacuum jug. The manipulator is controlled by a FSM that receives messages from a message service and controls the manipulator through another specialization of the Property Service. A camera is placed above the working area of the manipulator. An image analysis service extracts the positions of the vacuum jug and the coffee mugs from captured video frames. For this task, textbook computer vision methods including circular Hough transform and binary morphology are used. Since the movements of the manipulator do not need to be extremely accurate, the coordinate systems of the image plane and the manipulator are aligned with a simple affine transformation.

The coffee machine is started by sending it a message through the message service. The manipulator then grabs a mug the waitress robot has on its top, places it below the tip

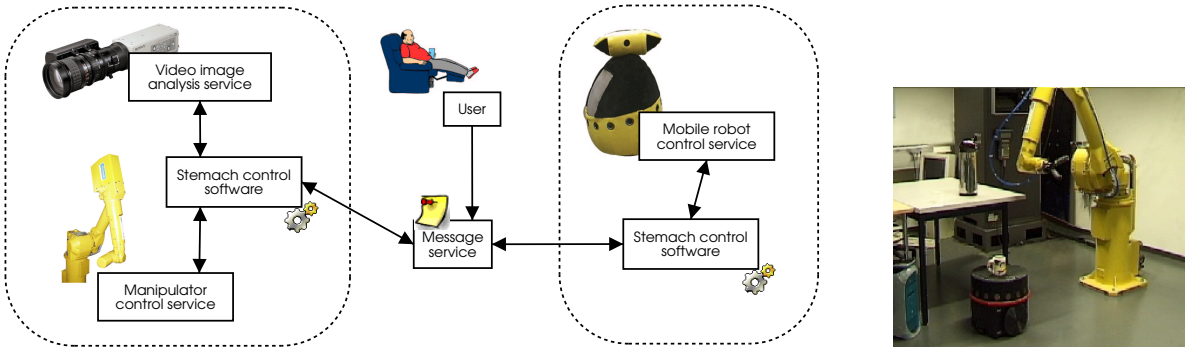

Fig. 2. Software architecture of the coffee serving system (a), and the system at work (b).

of the vacuum jug, pumps the button of the jug, and puts the mug back on the waitress. Finally, a message is sent that tells the waitress to leave. If the coffee machine cannot accomplish its task for some reason, an error message is sent.

The top-level state machine of the coffee machine is just like that of the simple example shown in Figure 1. A message from a message service fires an activation event, which drives the FSM to an "active" state. This state in turn is a more complex state machine that controls the movements of the manipulator. Once the coffee has been served, the automatic transition from "active" to "idle" is taken.

B. The Waitress

Due to the abstraction of behaviors, the waitress robot can be any mobile robot controllable through the Property Service interface. In the experiment reported in Section IV-C, the Super Scout 2 was used. To start the waitress, a message needs to be sent to it through the message service. The message can be formed in a number of ways, for example by pressing a button in a graphical user interface, with voice recognition, or with an e-mail message. The first two of these have already been implemented. Once the message is received, the waitress records its current position, drives to the working area of the manipulator, and sends a message to the coffee machine. It then waits for a confirmation and drives back to its starting position. The control software is a simple one-level FSM.

C. Experiment

To demonstrate the functioning of the architecture, a coffee-serving experiment was arranged. The action was initiated by sending a message to the Stemach software controlling the waitress by pressing a button in a user interface. With the aid of a virtual force field based path planner, the waitress successfully drove into the reach of the manipulator. Since the working area of the waitress was rather limited, the localization of the robot was simply measured with odometers. With a larger working area, other localization methods should be considered. Once the waitress arrived at its target position, it sent a message to the Stemach software controlling the coffee machine, which then successfully pumped coffee in the mug and returned it (Figure 2 (b)). Finally, the coffee machine sent a message to the waitress, which returned to its initial position.

V. Discussion and Conclusions

In this paper we presented a novel way of controlling complex distributed systems with a universal, easily extensible interface and a dynamically configurable state machine architecture. The architecture makes it possible to design complicated tasks on a high level without being concerned about the actual equipment that completes the required tasks. Due to the abstraction of tasks, programmers can use the same commands with all mobile robots, for example. Furthermore, the behavior of a system can be designed with a convenient graphical tool and altered while it is active. No deep knowledge on kinematics, networking technologies or robot programming is needed. Due to the lack of hard-coded control structures, the actual compiled controlling software be small and static, which allows it to be used in a wide variety of different applications. As a case study, the functioning of the system was successfully demonstrated in a coffee-serving application.

Currently, our implementation of the Stemach architecture is capable of executing very complex nested state machines. The PHP implementation does however have some shortcomings. Due to the nature of the PHP scripting language, it is not possible to run parallel actions on separate operating system threads. Instead, we use a scheduler that works on statement level rather than on processor instruction level. The advantage of this approach is that it makes it easy to design mutual exclusion mechanisms. The disadvantage is that fine-grained parallelism is not possible, and the capabilities of a CPU cannot be fully utilized. The external execution mode enables more sophisticated scheduling schemes, but none have been implemented yet.

The Property Service interface proved to be a very flexible way of controlling distributed services. As a next step, we are adding meta-data to the services that allows applications to automatically determine the types and functions of different properties. A simple self-configuring user interface for any distributed service has already been implemented.

The approach presented in this paper is directly applicable to many application areas. Context-aware systems are an example

of just the same kind of asynchronous, event-driven systems as the robotic system presented in this paper. One topic in our future research agenda will be applying this approach in developing intelligent, context-aware systems that contain a wide variety of distributed devices and software components communicating with each other.

ACKNOWLEDGEMENTS

The financial support provided by Academy of Finland is gratefully acknowledged.

REFERENCES

[1] E. Prassler, A. Ritter, C. Schaeffer, and P. Fiorini, "A short history of cleaning robots," *Autonomous Robots*, vol. 9, pp. 211–226, 2000.

[2] F. Hennie, *Finite-state models for logical machines*. New York: John Wiley & Sons, 1968.

[3] J. Pfeiffer, Jr., "Altaira: a rule-based visual language for small mobile robots," *Journal of Visual Languages and Computing*, vol. 9, pp. 127–150, 1998.

[4] R. Brooks, "A robust layered control system for a mobile robot," *IEEE Journal of Robotics and Automation*, vol. 2, no. 1, pp. 14–23, 1986.

[5] T. Henderson and E. Shilcrat, "Logical sensor systems," *Journal of Robotic Systems*, vol. 1, no. 2, pp. 169–193, 1984.

[6] J. Budenske and M. Gini, "Sensor explication: knowledge-based robotic plan execution through logical objects," *IEEE Transactions on Systems, Man, and Cybernetics, Part B*, vol. 27, no. 4, pp. 611–625, 1997.

[7] T. Henderson, C. Hansen, and B. Bhanu, "The specification of distributed sensing and control," *Journal of Robotic Systems*, vol. 2, no. 4, pp. 387–396, 1985.

[8] J. Wang, J. Su, and Y. Xi, "COM-based software architecture for multisensor fusion system," *Information Fusion*, vol. 2, no. 4, pp. 261–270, 2001.

[9] iRobot Corporation, "Mobility^{TM} software," http://www.irobot.com/rwi/p10.asp.

[10] OMG, *Notification Service Specification*. Object Management Group, 2002.

[11] J. Košecká, H. Christensen, and R. Bajcsy, "Experiments in behavior composition," *Robotics and Autonomous Systems*, vol. 19, pp. 287–298, 1997.

[12] Nohau Ltd., "Rhapsody UML design software," http://www.nohau.se/index_e.htm.

[13] P.-A. Muller, *Instant UML*. Wrox Press Ltd., 1997.

[14] K. Kim, Y. Matsusaka, and T. Kobayashi, "Inter-module cooperation architecture for interactive robot," in *Proc. Intl. Conf. on Intelligent Robots and Systems*, Lausanne, Switzerland, October 2002, pp. 2286–2291.

[15] D. Blank, J. Hudson, B. Mashburn, and E. Roberts, "The XRCL project: the university of Arkansas' entry into the AAAI 1999 mobile robot competition," University of Arkansas, Tech. Rep. Technical Report CSCE-1999-01, 1999.