A Software System for Robotic Learning by Experimentation

Iman Awaad, Ronny Hartanto, Beatriz Leon and Paul Plöger

Abstract— **The goal of the work presented here is to develop a robotic software system which enables robotic learning by experimentation within a distributed and heterogeneous setting. To meet this challenge, the authors specified, defined, developed, implemented and tested a component-based architecture called XPERSIF. The architecture comprises loosely-coupled, autonomous components that offer services through their well-defined interfaces and form a service-oriented architecture. The Ice middleware is used in the communication layer. Additionally, the successful integration of the XPERSim simulator into the system has enabled simultaneous quasi-realtime observation of the simulation by numerous, distributed users.**

I. INTRODUCTION

Software solutions have developed from a simple algorithm to programs that might contain more than one algorithm, to groups of programs forming an application. Nowadays, these solutions might encompass numerous applications running on a number of machines. More often, these applications are developed independently and must be integrated into a single architecture. Along with these developments, the complexity in the task of designing and abstracting (or architecting) these architectures has also grown. Principles that guide the structuring of such distributed applications are necessary, as is the use of technology which facilitates their development.

The test bed for the software system presented in this work is the XPERO project, the goal of which is robot learning by experimentation. The task at hand is the integration of required applications, such as planning of experiments, perception of parametrized features, robot motion control and knowledge-based learning, into a coherent cognitive architecture. This allows a mobile robot to use the methods involved in experimentation in order to learn about its environment. The software applications are distributed due to both the processing power needed and the multidisciplinary cooperation inherent in robotics research.

The results of this work demonstrate that the architecture is robust and flexible, and can be successfully scaled to facilitate the complete integration of the necessary applications, thus enabling robot learning by experimentation. The design supports composability, thus allowing components to be grouped together in order to provide an aggregate service. Distributed simulation enables real time tele-observation of the simulated experiment by users and applications.

I. Awaad, R. Hartanto and P. Plöger are with the Faculty of Autonomous Systems, Bonn-Rhein-Sieg University of Applied Sciences, Grantham Allee 20, 53757 Sankt Augustin, Germany ronny.hartanto@fh-bonn-rhein-sieg.de

B. Leon is with the Universitat Jaume I, Robotic Intelligence Laboratory, Castellon de la Plana, Spain

The following section will discuss and compare related work. Next, the necessary background information on the chosen approach is provided in the form of an overview of service-oriented architecture (SOA) and component-based software engineering (CBSE). The XPERSIF [1] system architecture and the component model which forms the basis for all components within the loop is then presented. An overview of the resulting architecture follows. As simulation was used from the start to speed up the pace of research [2] the novel solution used to distribute the simulation to numerous clients simultaneously is then presented. The results are presented in section VI along with a discussion of the work.

II. RELATED WORK

In this section, we present an overview of robotic software systems (RSS) and relate them to this work.

As laid out in [3], RSS tend to fall within one of three categories, driver and algorithm implementations, communication middleware, and robotic software frameworks. Often, the borders between these categories are blurred. Comparisons between RSS within different categories is misleading as each is motivated differently and serves a different function – i.e. they are simply unlike each other except in their shared goal of increasing reusability in robotics. An attempt is made here to present an example RSS from each of the three categories above, and relate them to this work.

The Player project [4] is an excellent example of the first type of RSS described above (driver and algorithm implementations). It includes a robot device interface which serves as a hardware abstraction layer (HAL) for robotic devices, as well as the robot simulators Stage and Gazebo. They are all open source and free. Player allows the same interface to be used to control the robot by providing ‘drivers’ which translate the abstract commands available in the interface into robot-specific commands.

Middleware for Robotics (Miro) is a distributed object oriented framework for mobile robot control, based on CORBA [5]. The overhead in terms of memory and processing power which results from the use of CORBA is a disadvantage of this solution. In addition, the complexity involved in understanding and learning to use it is also a hurdle. In comparison, the use of the much simpler and more efficient Ice middleware by XPERSIF precludes such problems. Miro is an example of the second type of RSS.

Orca [6] is very much a robotic software framework. It is an open-source framework for developing component-based robotic systems. It uses a CBSE approach which allows building-blocks (components) to be developed and

used together to create a more complex robotic system. The main motivation is the advancement of robotics research through the reuse of such building blocks. This is done by providing commonly-used interfaces, libraries and a repository of existing components. Orca's successor, Orca2, uses the Ice middleware.

While the use of the Player project would address the issue of robot control and perhaps simulation, it could not be used for the integration of a complete robotic system (such as the cognitive architecture presented here). The same can be said of Miro. Of the three RSS presented above, Orca is most similar to XPERSIF in that it is also a thin framework which utilizes the Ice middleware, provides a simple component model and uses simple and efficient communications patterns. An added advantage of the XPERSIF framework is its service-oriented nature. Despite being a mature project, Orca's repository does not provide interfaces, components or libraries which offer the more advanced functionalities relating to cognitive architectures. Section VI presents the results of the XPERSIF framework and architecture which highlight the advantages of its design.

III. APPROCH

A CBSE approach has been used to encapsulate the functionalities of the robotic software and hardware systems into components. These components have been loosely coupled to form a SOA.

'Service-orientation' is a design paradigm for architecting a distributed architecture which centers around loosely-coupled autonomous components and groups of components providing services in order to carry out a given task. The principles of this paradigm allow the separation of the business and application logic domains of an enterprise. They guide their structuring into the layers associated with a SOA model. Additionally, they provide tennets for the granularity and other characteristics towards which individual services should strive. SOAs are basically a collection of services. A service is well-defined and self-contained, and does not depend on the context or state of other services [7]. These services can be registered with a central registry which allows service requesters to discover them. This is the essence – the abstract idea of a SOA.

How a SOA is implemented can vary. An implementation approach based on that of traditional distributed architectures, or alternatively one utilizing 'web services' (WS) which takes advantage of the Internet and its proprietary-free communications network may be adopted. Such a SOA might use document-style messages which are encoded in protocols such as the Simple Object Access Protocol (SOAP) to transport, process and route the payload which itself is represented in the Xtensible Markup Language (XML).

In this work, the traditional distributed architecture is used in the implementation of a SOA. Additionally, no workflow and orchestration (in the strict SOA sense) are used. These are not necessary as the architecture presented here has no need for service discoverability and the use of services within the architecture is static rather than dynamic. The composability of SOA is achieved through the use of the CBSE approach. CBSE adopts the doctrine of 'divide and conquer' by breaking down a system into functional or logical components. These components (and the services they provide) are accessed through their interfaces. The component model used for this work is defined in the following section.

This makes it possible for the actual implementation details (algorithms, etc.) of a component to be hidden behind its interface, thus enabling plug and play functionality through shared interfaces for components using varying implementations to provide the same service. These modular and reusable components can be coupled and layered to build larger systems which are robust and easy to maintain. Their development and testing is simpler, as each component is developed independently from other components (while sharing a common component model). This characteristic means that an expert in each component's functionality can develop the component independently based on a component model without needing to possess expertise in all issues relating to the system as a whole. This makes the development time shorter (as work can proceed in parallel) than that needed for non-CB systems.

Components can be placed on various computers – dedicated application servers – to form a distributed architecture. Traditionally, Remote Procedure Calls (RPCs) are used for communication between components within such a distributed architecture. This is a point of difference between SOAs based on these distributed architectures and those based on WS, as the communication between services in the latter case is accomplished through document-style messages using protocols such as SOAP, which are as self-sufficient as possible (containing policy rules, meta-information and processing instructions for example). This tends to result in larger messages that are sent and received less frequently in comparison to RPC communication as used in traditional distributed architectures.

IV. ARCHITECTURE

A component model provides a standard for developing components within a framework. This facilitates the development process. In this section, we present the model and the architecture. Components within the XPERSIF architecture are classified into one of three basic groups of components, namely *basic* components, *organizational components* responsible for managing a hierarchy of components, and *aggregate components* which are organizational components which appear as a single component but are composed of individual components which cooperate to provide the services of the aggregate component by using the facade software pattern.

A component's structure can be summarized by stating that it offers services as commands and operations (as defined below) and notifies its users of the final state of the service (Fig. 1). In order to provide this functionality several abstract interfaces are specified. Namely, the Operation interface (which provides functionality for component administration),

the Subject interface (which provides a means to subscribe to notifications) and the Observer interface (which provides a means to receive notifications). The component-specific functionality is specified within the Component interface. As none of the components within the system are hard real-time components, this component model meets the soft real-time requirements in the simplest way possible. Operations are used for services which complete quickly, commands for those that need more time, and notifications serve to enable a call to a command to quickly return (thus enabling a non-blocking call) and to provide a monitoring and error handling mechanism. These three interfaces are sufficient for the requirements of the project. To prevent components from blocking, while at the same time allowing for tasks which may have varying duration, a differentiation is made between commands and operations. Both commands and operations return immediately, however, commands will start an asynchronous process which completes when the component notifies the original caller of the command's completion. Commands are used for tasks which may take time to complete, such as moving to a position for example or gripping an object. They are implemented as non-blocking RPCs. From a planning perspective, commands could be seen as planning operators and as such should have preconditions and effects so that a planner can make use of this information. These preconditions and effects may be seen as a contract. Operations, on the other hand, do not start an asynchronous process. Tasks such as getting the readings from the IR sensors or setting the maximum velocity are implemented as operations, as they take very little time to execute and pose no risk of blocking the component when they are called. This distinction has many effects on the architecture. It is useful to specify preconditions and effects for operations as well. For example, an operation which should deliver the shape of an object might specify as a precondition that the object is in view at a specified distance from the camera. The definition of such preconditions for operations would then form a contract (as with commands).

A. The XPERSIF components

A component diagram depicting the data flow between the various components of the loop is shown in Fig. 2. The components here have been grouped according to their functionality. This diagram does not elaborate on the implementation of each of these components – for example, it does not mention which are applications and which are components that are grouped to form an application. They simply show the components of the loop and the data flowing between them. The LOOPMODEL component which is the organizational center of the loop is seen at the center of Fig. 2. It serves as an entry point to the loop for a graphical user interface (GUI) (or a console client as in the current implementation) which uses it to configure the loop. It is responsible for parametrizing, starting, monitoring and exiting the necessary components within the loop. For the sake of simplicity, the flow of the component status information from each component to the LOOPMODEL is not depicted.

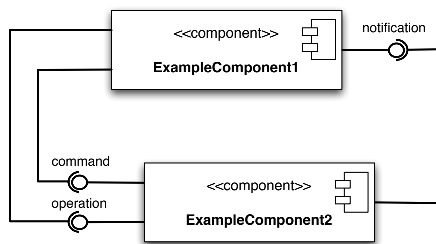

Fig. 1. The XPERSIF component model

The DESIGNOFEXPERIMENTS component is responsible for designing effective experiments when the robot is in the experimentation state. The GOALDESIGN module is responsible for the robot's actions when it is not in the experimentation state. Once a goal state for the robot has been formulated, the planner should produce a plan to achieve this goal, and the EXECUTION component should then execute this plan.

The robotic embodiment is itself represented by a group of components. As the embodiments are mobile manipulators, the components include a RELOCATION and a MANIPULATION component. In addition, a PERCEPTION component provides access to the embodiment's sensors. These components receive commands from the execution component and return monitoring information to it. A central point to query the embodiment's state is the ROBOTMODEL component. It is part of the organizational layer as it is responsible for starting those components which make up the robotic embodiment. These components make API calls to either the physical embodiment or to the simulated embodiment within the XPERSim simulator, thus, it may be seen as a form of tool-driven validation.

An overhead camera is often used within the cognitive

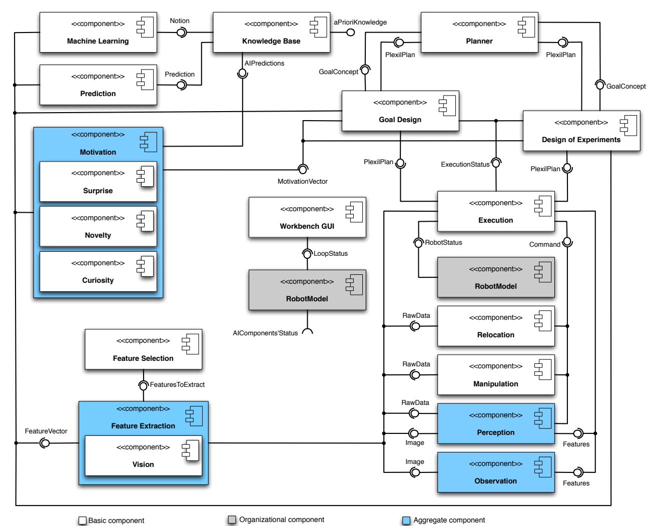

Fig. 2. A component diagram showing the data flow between XPERSIF components

loop both by the human researcher to tele-observe the experiment, and by the robot itself to provide ground truth. In the latter case, the view from the overhead camera is provided to the robot as a service (by the OBSERVATION component). Both this component and the PERCEPTION component are displayed as aggregate components in Fig. 2 as they use their own instances of the ROBOTFEATUREEXTRACTION meta-component. They are responsible for initializing the instances of this component and are the sole users of the interfaces.

The FEATUREEXTRACTION component takes as input the raw sensor data from the embodiment and the overhead camera, and extracts meaningful features (by default objects and their poses). Should the FEATURESELECTION component specify additional features to be extracted, this may be carried out by either the FEATUREEXTRACTION component itself or the VISION component. The FEATUREEXTRACTION component generates a feature vector as the output of this process.

Components such as those of the MOTIVATION aggregate component and the MACHINELEARNING component receive these feature vectors and use them to generate their own outputs (curiosity and surprise values, a prediction, a notion, etc). For the sake of simplicity, the diagram in Fig. 2 shows only the data flowing into the knowledge base, although it serves as a central point for all components to query for data at anytime.

V. DISTRIBUTING THE SIMULATION

The integration of the XPERSim simulator provided not only the means to conduct research but a means to validate the software system itself. This section details the efforts made to distribute the simulation for tele-observation by researchers and for processing by individual components. Although the implementation is specific to the XPERSim simulator, the same approach can conceptually be used for other simulators. This integration of XPERSim into XPERSIF allows the simulation to be run in a distributed setting.

XPERSim is a 3D simulator based on open source components, built by the authors, that quickly and easily constructs an accurate and photo-realistic simulation for robots of arbitrary morphology and of the environments within which they function. XPERSim achieves such high quality visualization by using the Object-Oriented Graphics Rendering Engine 3D (Ogre) engine to render the simulation whose dynamics are calculated by Open Dynamics Engine (ODE). Simultaneous multiple camera simulation of the rendered scene is possible at high frame rates. A library of sensors and actuators commonly-used in robotics is available.

While tele-operation and tele-observation of the simulation were previously implemented, the solution for tele-observation was provided with a focus on fulfilling a use case for data generation which provided traces for the machine learning tools. These traces included the simulated image which was requested and transmitted through a synchronous RPC call. While this requirement was met, the solution did not enable a frame-by-frame view of the simulation. The specification of new use cases specified the need for the architecture to supply quasi-real time observation of the simulated image. The implementation of the solution is presented in this section. A number of issues precluded the use of the same method for true real time tele-observation of the experiment. The first of these is the presence of a bottleneck in obtaining the rendered image from the GPU (Graphics Processing Unit) to the CPU which makes the process of simply obtaining the image a time-consuming affair. The second issue is the transmission of the image itself which takes time. It should be noted that these issues made infeasible the real time or quasi-real time tele-observation of the experiment by even one single client. In order to facilitate scalability, bottlenecks must be avoided.

The solution presented here uses a proven methodology (often implemented in multi-player games) which involves moving the rendering of images from the server-side to the client-side by sending out a subset of the scene information to ensure that all clients are operating synchronously [8]. This drastically reduces the amount of data being transmitted and is possible due to the scene-oriented nature of the XPERSim simulation. As mentioned previously, the Ogre 3D rendering engine uses scene-graphs to represent hierarchies, which simplifies the processing of objects or groups of objects. A scene-graph consists of nodes (with parent nodes and child nodes). If a parent node is translated or rotated, this transformation is applied to the child scene nodes as well.

The latency resulting from the distributed nature of the application is ameliorated by sending the node information from the simulator while the client is rendering the previous one – i.e. the server does not wait for the client to request the image but sends it continuously once it has subscribed. The method described above to distribute a simulation to multiple clients is implemented here by decoupling the physics and graphics engines from XPERSim to create an XPERSim Server (calculating dynamics) and a TeleSimView client (rendering the nodes at their new positions). The XPERSim Server sends out the positions and orientations of all scene-nodes to the clients that simply transform the specified nodes to the specified positions and orientations and in so doing produce the same scene in an efficient and real time manner.

As mentioned above, the XPERSim Server is now solely responsible for calculating the dynamics of the simulation and for their distribution to the clients. The separation of the two engines was straightforward due to the modular structure of the simulator. Various methods for transmitting the node information were evaluated. During the start-up of the simulation and the creation of the ODE bodies, the information pertaining to the Ogre-scene is accumulated. This information is stored in a container structure which is requested by the CAMERA subcomponents which will publish the images. The same scene is rendered from each camera's position. As the robot's camera is attached to it, it will automatically be moved when the robot moves. If a pan/tilt camera is used, then its position and orientation could be sent out as a node.

In an effort to further reduce network latency, a one-way invocation is used to send the new frame. This can in fact be

quite expensive when many such small messages need to be sent. This is because the run time taps into the OS kernel for each message and because each of these messages is sent out with its own message header [9]. To ameliorate this problem, batched one-way invocations are used. This allows the Ice run time to buffer these small messages until the XPERSim Server explicitly flushes them.

The TeleSimView client is used to view the simulated scene. With the same node information, the view from both cameras is rendered. The subscription to receive the node information is made with the XPERSIF components: PERCEPTION (robot camera) or OBSERVATION (overhead camera). The client only requires Ogre (and its dependencies). Ogre itself has always been cross-platform compatible. With the release of Ogre version 1.4.6 (a.k.a. 'Eihort'), the same source code may be used across platforms with the use of the Object-oriented Input System (OIS) platform.

A two-way invocation to the PERCEPTION (or OBSERVATION) component fetches the scene which will be created and subsequently updated. The creation of the scene involves the creation and attaching of nodes, their positioning and the creation of such basic scene items as the plane, lights, and sky. Once this has been done, the client uses operations found within the interfaces which are extended by the PERCEPTION and OBSERVATION components in order to subscribe as an image-observer. As soon as this is done, the images will be transmitted to it from the relevant camera subcomponent (the image-provider).

VI. RESULTS AND DISCUSSION

The XPERO project has provided XPERSIF with an invaluable testing ground. The evaluation of the software integration framework is measured here qualitatively against various criteria such as flexibility, reusability, scalability, ease of use, level of documentation, and development time. After this evaluation, empirical results for the distributed simulation solution are presented.

Flexibility of the framework was the core criterion in the development process. It is manifested not only by the ease of changing implementations independently of the abstract interfaces (e.g. using different planning algorithms beneath the same interface) but also by the ease with which components use each other's services. Flexibility was enhanced by the use of the organizational components which centralize the point at which changes might need to be made. Even a change of interfaces would be simple enough to propagate. The flexibility was further enhanced through the adherence to the SOA principle of service-stateless. The services offered through the component interfaces are at a level of granularity which enables their use under different control flow scenarios which have been specified within the project.

Reusability has been achieved by allowing various existing implementations of an application to be reused beneath the interfaces. Numerous instances of components (such as VISION) also contributed to the reusability of a component.

Scalability has been ensured through the use of the component-based approach and through the consistent use of simple and efficient communication patterns.

Ease of development was facilitated by the use of a single basic component model (augmented as need be for organizational components). The simplicity in design of this basic component model, which is nonetheless robust and efficient in propagating information through the various layers, is an achievement in itself. It allowed the system to be easily debugged and facilitated error handling which results in a *robust* system. The implemented base applications also contributed to the ease of integration of components and applications. This ease of development also contributed to the *extensibility* of the architecture. The framework was designed with future needs in mind (e.g. a workbench for robot learning by experimentation, the use of stereo vision, of multiple robots, etc). This includes the facilitation of implementing a change in the experimentation process. The use of the GOALDESIGN and DESIGNOFEXPERIMENTS modules to orchestrate the loop allows such changes to be confined to these components. The interfaces of the remaining components are abstract enough to not need amendments.

The level of *documentation* is accurate and consistent at a variety of levels including the source code, installation and user guides for the various versions which have so far been released. This holds for both the XPERSim and the XPERSIF projects. The Ice middleware too, has a high level of documentation which is both extensive and easy to reference.

Sliding autonomy [10] is, in the case of this work, a valuable criterion, as the evaluation of the functional performance of the architecture must be carried out from the viewpoint of both the researcher and the robot (the two XPERSIF users), and often must be carried out from both points of view simultaneously. The ability to use XPERSIF under varying levels of autonomy is a necessity. The use of the LOOPMODEL component to parametrize the experiment and the enabling of placeholders for the application (e.g. allowing a pre-generated plan to be used through the interface) provided this varying degree of autonomy.

Distribution of the simulation through the integration of XPERSim into the XPERSIF architecture was successfully achieved. The scalability of the implementation described above was evaluated by measuring the impact on the quality of the simulation by varying the number of subscribers to the tele-observation service. The initial efforts to distribute the simulation provided the image's color pixel values, in the BGR format, as a sequence of integers. In addition to the image itself, the width and height of the image, as well as the time to which it belongs, were also sent. A set of experimental evaluations was carried out to measure the time in seconds which is needed to receive a new image of size 416 x 600 pixels. For this set of evaluations, the image was sent from a Pentium 4.3GHz computer with 2 GB RAM and an ATI Radeon X800 GTO to a machine equipped with an AMD Turion 64 Mobile 1.79 GHz processor, 1GB of RAM and an ATI Radeon Express 200M. The round trip time needed to deliver the image was measured at 9.5346 seconds (i.e. at a frame rate of <1 fps). Using the approach

TABLE I

THE TIME IN SECONDS BETWEEN RECEIVING TWO SUBSEQUENT IMAGES USING THE BATCHED ONE-WAY INVOCATION METHOD

Trial	1 client	3 clients	5 clients	10 clients
1	0.0039 (s)	0.0039 (s)	0.0219 (s)	0.0227 (s)
2	0.0023 (s)	0.0172 (s)	0.0128 (s)	0.0352 (s)
3	0.0075 (s)	0.0036 (s)	0.0120 (s)	0.0448 (s)
Mean	0.0046 (s)	0.0082 (s)	0.0156 (s)	0.0342 (s)

described above, the time measured between receiving two subsequent images was 0.0046 seconds (i.e. 217 fps).

Table I shows the measurements made when one, three, five and then ten clients are subscribed to the service. For this set of experiments, XPERSim Server was running on the first machine mentioned above, while the clients were distributed across three additional computers with network speeds of 18-24 Mbps. All experiments were repeated three times, measuring the time it took for 60 frames to be delivered to the TeleSimView client. It is worth noting that the size of the image to be rendered is inconsequential. As nodes are being sent and not an image, it is the number of nodes within a scene that affects the time and not the image size. For the test case above, 15 nodes were transmitted (representing the Khepera robot and the four cubes). Using this information, the scene may be rendered from the viewpoint of any number of cameras.

VII. CONCLUSIONS AND FUTURE WORK

A. Conclusions

The resulting software architecture easily and efficiently allows the integration of components (both software and hardware) which run on heterogeneous platforms and languages. The use of CBSE allows the software architecture to maximize concurrency in the application development process of the various research groups. The adoption of the SOA approach in the design of the framework has produced a system which is highly flexible and maintainable. The framework is data-centric with communication of the data playing a significant role in the design. The simplicity of the communication patterns contributes to the efficiency and flexibility of the framework. The data itself which is exchanged between components is abstracted in such a way as to maintain interfaces which are as simple as possible. Additionally, the solution for the tele-observation of experiments is a significant contribution to the framework as a whole. The architecture thus developed has successfully enabled effective, simultaneous, quasi-realtime observation of the simulation by numerous, distributed users.

B. Future Work

A number of issues are currently being addressed. In the current implementation of the XPERSim simulator, no distinction is being made between parent nodes and child nodes. It is recommended however that this distinction be made as it would reduce the number of nodes whose data needs to be transmitted (transmit parent nodes only and nodes which can be moved separately from the hierarchy – a gripper for example which, despite being a member of the robot node, may be moved on its own). Additionally, the TeleSimView client is being upgraded to use the latest version of Ogre.

VIII. ACKNOWLEDGMENTS

The authors express their gratitude to Karl-Heinz Sylla for his guidance and to the researchers of the XPERO project for their feedback and support. Many thanks to Keyan Ghazi-Zahedi for his most useful comments.

The work described in this article has been partially funded by the European Commission's Sixth Framework Programme under contract no. 029427 as part of the Specific Targeted Research Project XPERO ("Robotic Learning by Experimentation").

REFERENCES

[1] I. Awaad and B. Leon, *XPERSIF: A Component-based Software Integration Framework for Robotic Learning by Experimentation*, Master Thesis, Bonn-Rhein-Sieg University of Applied Sciences, March 2008.

[2] I. Bratko, "Initial Experiments in Robot Discovery in XPERO", *in ICRA 2007 Workshop on "Concept Learning for Embedded Agents"*, April 2007.

[3] A. Makarenko, A. Brooks, and T. Kaupp, "On the Benefits of Making Robotic Software Frameworks Thin", *International Conference on Intelligent Robots and Systems Workshop*, October 2007.

[4] B. Gerkey and contributors, *The Player Robot Device Interface*, 2005.

[5] T. M. Group, Miro Manual Version 0.9.4, January 2006.

[6] A. Makarenko, *The ORCA Manual 2.7.0*, October 2007.

[7] T. Erl, *Serive-Oriented Architecture: Concepts, Technology, and Design*, Prentice Hall, Upper Saddle River, NJ; 2005.

[8] O. Forums, "Streaming a scene rendered by a camera", Online at http://www.ogre3d.org/phpBB2/viewtopic.php?p=224646, May 2007.

[9] M. Henning and M. Spruiell ed., *Distributed Programming with Ice revision 3.2*, ZeroC Inc., March 2007.

[10] J. D. Brookshire, *Enhancing multi-robot coordinated teams with sliding autonomy*, Masters thesis, Carnegie Mellon University, 2004.