An Extensible System Architecture for Cooperative Mobile Robots

Falko Dressler and Mesut Ipek

Computer Networks and Communication Systems
University of Erlangen-Nürnberg, Germany

Abstract. Self-organization in autonomous sensor/actuator networks is an emerging research area. Special focus lies on cooperation issues of mobile robot systems. We employed multiple software engineering methodologies to develop an easily extensible core system named robrain. In this paper, we present the structure of our plugin-based system architecture and demonstrate its applicability based on well-defined key requirements. The primary application area is the development of extensible software systems for teams of collaborating mobile robots. Using available extensions, the system can be interconnected with sensor networks or entities controlling the behavior of a multi-robot system. We already used this system in several courses at our university. We have discovered that several studies can be easily conducted due to the modular structure and the provided remote controllability.

1 Introduction

Robotics and Automation are key technologies that focus on challenging future applications. They demand for new methods and solutions in various research areas. Efficient utilization of available resources is one of the most important issues. This applies to single robots as well as to multi-robot systems. In this context, collaborative task management and synchronization and coordination between autonomous robot systems is of major interest. In this context, new research lines appear concerning the development of mobile sensor/actuator networks.

Therefore, robot systems are the core of variety areas of research. In this paper, we address software engineering issues for mobile robot systems that have to deal with frequent hardware changes, support for multiple platforms, and operation in different working environment. Besides research aspects that demand for an easily extensible software and hardware architecture, the possibility to employ the same robot systems for educational purposes is a key requirement. Therefore, rapid prototyping of new drivers, functions, and even behaviors must be supported by the core system software. Additionally, there is also the need for cooperation with other units in the environment such human operators or other robots.

Hence, the need for an easy extensible control system requiring few programming effort in order to include new functionality was born. The aim is to allow to merge an existing stand alone application into an existing control system.

A attempt to fulfill these requirements will be explained in the following. The rest of the paper if organized as follows. Section II describes the goals of our current research work. Section III depicts the system design including the key requirements and software engineering issues. Then, in section IV, the extensible robot control system named robrain is discussed. Finally, the paper is summarized by some conclusions.

2 ROSES - Robot assisted sensor networks

The development and the control of self-organizing, self-configuring, self-healing, self-managing, and adaptive communication systems and networks showing an emergent behavior are the primary research aspects of the ROSES project [1]. The mentioned aspects are studied in the area of autonomous sensor/actuator networks, i.e. a combination of mobile robot systems and stationary sensor networks. The introduction of mobility as well as the limited resources of typical sensor nodes leads to new problems, challenges, and solution spaces in terms of efficient data management and communication. Here, it is distinguished between sensor assisted teams of robots and robot assisted mobile sensor networks. The former means that robots might use the sensor network for more accurate localization and navigation or as an infrastructure for successful communication. The latter means the employment of robot systems for maintenance in sensor network or for providing communication relays.

Research Goals

- Energy efficient operation, communication, and navigation
- Sensor network assisted localization and navigation of the robots
- Utilization of the robots as a communication relay between a sensor network and a global network, e.g. the Internet
- Quality of service aware communication in heterogeneous mobile networks with dynamic topology
- Optimized task allocation and communication based on application and energy constraints
- Secure communication and data management in mobile sensor networks

In order to address these objectives, novel models and methodologies are investigated for energy and application aware communication [2], different localization techniques for optimized high-precision navigation are combined, mobile robots and stationary sensor nodes are integrated to autonomous sensor/actuator networks, and research on bio-inspired communication methods is conducted. The lab environment includes the Robertino robot[1] platform as well as the Mica2 and BTnode sensor motes running TinyOS. One primary application scenario is the exploration and monitoring of unknown surroundings.

[1] http://www.openrobertino.org

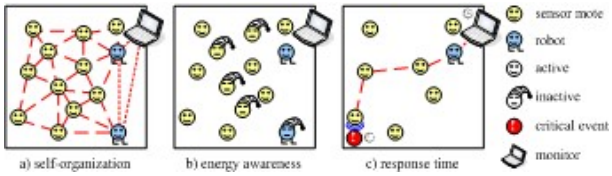

Fig. 1. Challenges and research directions in the area of mobile sensor/actuator networks

In figure 1, the most challenging issues are depicted. Self-organization (a) means, that global tasks are solved without the need of a central control or management, i.e. the network notes explore their neighborhood and distribute the decomposed task in an appropriate way. Energy awareness (b) is a key property of the systems in focus. Sensor networks should be available over a long time period without external maintenance. Finally, the response time (c) of the system, i.e. the time between the occurrence of an event and the appropriate response, is an important aspect.

Current Activities The mobile robot systems are equiped with a modular control system allowing them to act completely autonomous. In order to achieve this goal, modules for accessing the sensor facilities, for movement, localization and navigation, and task allocation are work in progress. Mostly finished is the energy control module composed of a battery management and the corresponding characteristics. In collaboration with the Fraunhofer Institute for Integrated Circuits IIS, a special circuit for voltage and current control was developed [3]. The energy module allows an approximation of the remaining energy and an estimation of energy requirement of forthcoming tasks. Additionally, the connection to the sensor network is provided by another module used for more precise localization techniques [4]. In the field of sensor networks, ad hoc routing algorithms are evaluated with the focus on energy constraints and timeliness of the communications. To achieve this goal, "real" sensor motes are interconnected with simulation models to achieve more accurate results than available in previous simulations.

3 System Design

In this section, the primary requirements are discussed that directed the design and the implementation as well as some important design issues concerning the extensibility, the adaptability, and the simplified usage in research and education.

3.1 Requirements

Before investigating the developed system in more detail, the design issues that lead to the proposed architecture should be discussed. During the development, the following system requirements have been identified:

Extensibility An easy extensible architecture is needed with the ability to abstraction to evolve higher problem areas (e.g. provide hardware layer abstraction using existing device driver).

Adaptability The adaptability on hardware changes to include only those parts of extensions which are actually needed to perform tasks using existing hardware and exclude those functions which are not needed.

Ease of Integration The easy integration of existing software to the system to extend the systems functionality and/or to provide an easy communication layer between different standalone applications.

Applicability for Education On of the most important requirements is the applicability of the architecture for educational and research purposes.

To adopt these requirements to the presented architecture, we use the mechanism of a extensible architecture where each extension component can be added or removed at run time. In the following, we use the words *extension* and *plugin* equivalently.

3.2 Core System

To fulfill the requirements of an easy extensible architecture and the ability to extend the system by incorporating already written software a core of three classes evolved as shown in figure 2 (marked yellow).

The core consists of three parts: the *PluginManager*, a *PluginTag*, and an associated *Plugin* interface.

PluginManager The *PluginManager* holds information about available extensions and which correlations exist in between them, i.e. which extensions are used by other extensions. It also propagates the access calls to the appropriate plugintag and controls the allowed access behavior.

At the beginning it determines the plugin places and generates for each plugin a plugintag and leads the plugintag to load the plugin and set up his environment.

So the *PluginManager* performs as a link between the *PluginTags*.

PluginTag A *PluginTag* is responsible for a specific plugin. Its tasks are to provide dependability information concerning other plugins, to privide access control, and to forward requests to other plugins to the pluginmanager. Temporarily blocked access to a plugin in case of an extension that can only be accessed once at a time can also checked by the *PluginTag*. Further tasks could be easily integrated.

During an initial start sequence, it loads the plugin into the system and initiates an initialization procedure. It also reads the parameter values from a

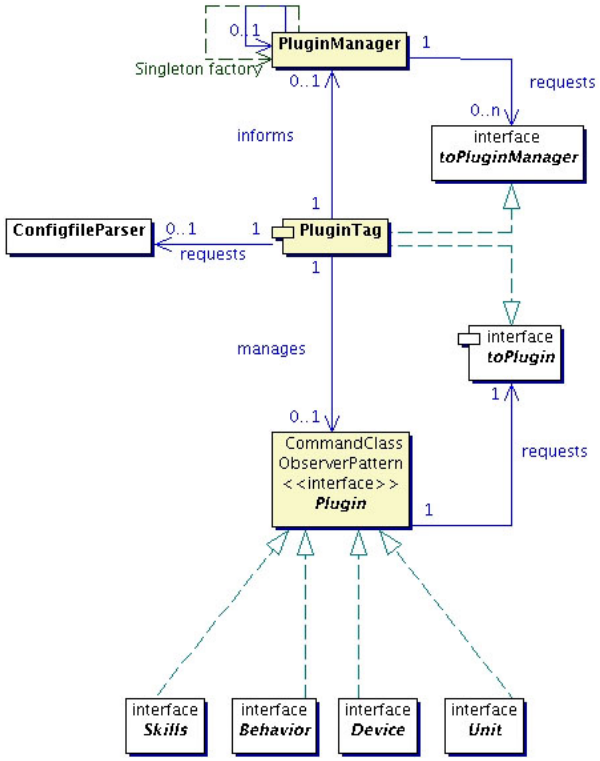

Fig. 2. The core functionality of the plugin system

configuration file and provides an environment for the plugin, which holds a map of the parameters, the plugin name, and the interface to the system. All requests of the plugin where meant to be going through the plugintag expect the communication with other plugins after a first successful plugin access. This behavior is needed due security reasons: no plugin can actually forge its name to gain access to a plugin to which it has no access.

Therefore, the *PluginTag* performs the management role of a plugin and works as a middleware between the plugin and pluginmanager.

Plugin Interface The *Plugin Interface* provides the necessary functions which have to be minimally implemented in order to init, activate, and deactivate the plugin properly.

The following functions have to be implemented:

– The *init* function includes all initialization parameters at the beginning. They are usually located in the constructor of a class.

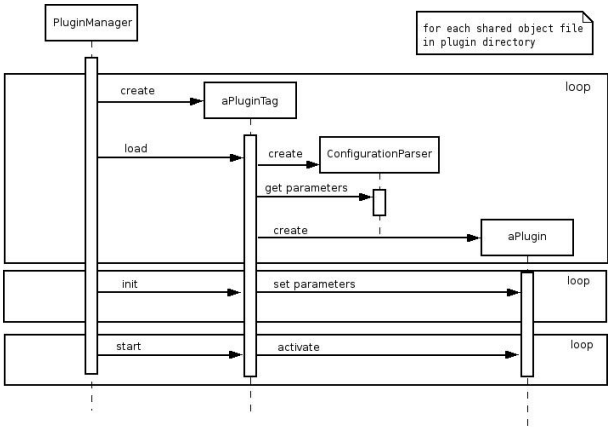

Fig. 3. Start sequence of the system including involved modules and according functions

- The *activate* function controls necessary relationships to other plugins. If a plugin which requires to access functionality provided by other plugins, the access requests are forwarded to the according plugins.
- The *deactivate* function includes cleaning operations like the destructor of a class.
- The *emergencyHalt* function includes all necessary functionality in order to immediately stop an existing task of the plugin.

Additional functions have to be added for a particular functionality of the extension to make it meaningful. The complete initialization procedure is depicted in figure 3.

3.3 Plugin System

Two kind of extensions to the core system exist. Active plugins have to be running all the time since system startup, e.g. to collect information from dedicated hardware modules. Usually, they use their own thread and start always a control thread in the system. The second possibility is that an extension is passive. Such plugins work like a common function call, i.e. they become active during a function call, process the demanded information, and return the outcome. They actually do not start a thread of their own but become activated by a control thread.

Besides the active and passive behavior of the extensions, we derive from the Plugin interface multiple types of interfaces. They are set in a hierarchical relationship to each other. Currently, four types are available that will be described in the following. Figure 4 shows an example session from the scope of hierarchical structure, the cooperation and active and passive extensions.

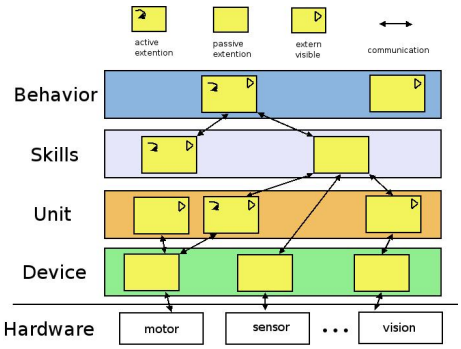

Fig. 4. An example of cooperating extensions

Device *Device* type extensions directly access a specific hardware module. Therefore, they provide functionality dedicated to this part of the hardware. This may be also a virtual hardware like the common plugin described in section *Robrain*.

Unit *Unit* type extensions are based on Device type extensions and provide a higher abstraction levels of the provided functionality. Examples are a tcp socket connection extension and an interactive shell extension that uses the command plugin in order to call extension functions through a socket connection and command line, respectively.

Skills *Skills* type extensions may use Unit type extensions or provide a skill that does not rely on any hardware layer abstraction. Such a plugin could be a counter which counts the seconds still start of the system.

Behavior *Behavior* type extensions are using all other extension types to build a behavior of the overall system. An example could be the monitoring of a room using the driver unit, measuring the distance, and getting snapshots from the vision control.

The division into hierarchical types makes it possible to design complex tasks that the robot can provide, like observing a room as a *Behavior* extension. Such an extension uses a *Unit* plugin which currently grabs a picture from the camera using the vision control *Device* plugin. Obviously, the functionality of the extensions becomes more abstract with a higher application type.

Note that if a extension has to use some functionality of an extensions at the same level, the current system design demands that these two extensions have to be merged together.

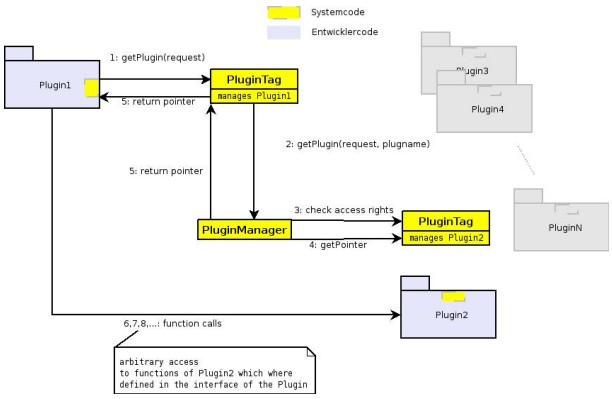

Fig. 5. Internal access request by a plugin

3.4 Access Control and Parameters

Internal access to plugins Each extension is described by an interface to the system. The interface is described by an interface class with according function declarations. The aim is to separate the internal implementation of the extension from the overall system. Each extension is known to the system by his interface header file. Only those functions which are described in this header file will be accessed internally from the system. Another advantage is that an extension could be delivered without the source code. The correlations between the core system and the plugin for internal access are depicted in figure 5.

External access to plugins Some extensions should be accessed from outside the system. These provide functionality through different interfaces like tcp socket, command line, or even through speech recognition.

The view to the plugin is different from the internal access. It is possible to define which functions of the plugin should be seen from outside in a file like the interface file. This could be somehow disjunctive to the functions from internal view or be the same. It must be taken into consideration that a return value of pointer type or an array do not make much sense since the external access control function converts all return parameters to a string.

Parameter passing Because each function has a different function signature caused by an unknown number of arguments, argument types, and return values. Therefore, each function will be wrapped the using the following approach.

Figure 6 shows three classes: the input and output arguments will be collapsed into derived classes of CommandInput and CommandOutput, respectively. The input class holds the input parameters of the function and the output class holds the return value of the function. In a derived class of the CommandClass the function will be called by his CommandInput class and the return value will be stored in the CommandOutput class of the function.

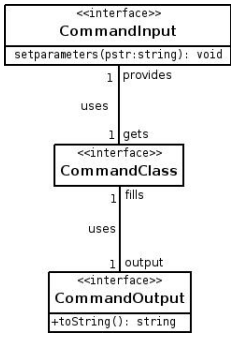

Fig. 6. Abstract wrapper classes for input and output parameters of a function

All functions of an extension will be wrapped by a dispatcher which calls the appropriate function from the list of available functions. Each function and the appropriate wrapper classes are identified by a signature which composed by its return value, its function name, and its type of input parameters. For example, a function *void foo(int a, string b)* will have the signature *void_foo_int_string*. Note that this permits the possibility of using function overloading as provided by C++. The derived CommandInput class would be named *CommonInput_void_foo_int_string* and so on.

As a result, a dispatcher functions is created that returns the appropriate wrapper class as recognized by the function signature. Since CommandInput and CommandOutput classes have always the same signature, any arbitrary function can be collapsed into a black box scheme. In this case, the types of parameters must not be known by the command plugin, which actually calls the function by setting the argument string with *setparameters(argumentstring)* and collecting the return value as a string with *toString()*.

The wrapper functions, input and output classes are automatically generated by a python script that is using the external interface description.

Notification System To be able to notify extensions about a finished task or other events, we provide a notification system. Here, we use the Observer Pattern[2]. A slightly modified version which allows to attach arbitrary functions with a well defined signature allows to attach different handling for different tasks.

Exclusive Access Some extensions can not be used by more than one plugin at the same time. This can be defined by a value in the configuration file. It such case, the PluginTag prohibits the access request and the PluginManager returns the name of the plugin which currently uses the plugin to the requester. Note that external access to such extensions does not make sense since the functionality provided by this kind of extensions located in the lower levels of abstraction.

[2] http://www.dofactory.com/Patterns/PatternObserver.aspx

4 Robrain

An implementation based on the described system design was done in C++ and is called *robrain*. In this section, we provide some more detailed information about the implementation and possible extensions.

4.1 Programming Environment

C++ / Common C++ We decided to use C++ for the implementation. There were two reasons for this decision: on the one hand, the object-oriented system design demanded for an appropriate programming language and on the other hand, languages like JAVA required too much resources on the embedded systems controlling our robot systems. We used Common C++[3] to provide the threading functionality. The command line options parsing and socket handling also made by using the library functions.

The plugin skeleton A shell script is used to generate an extension skeleton. Three parameters are needed by the script: the name and the type of the plugin and its visibility. The plugin can be hidden for external access or it can be visible. For the latter option the declarations of the wrapper class and the dispatcher function as well as the appropriate include files will ge generated later for further usage.

Configuration file and the interface headers Each plugin is configured by an individual configuration file, which has three sections. One for the plugin internal parameters which can be arbitrarily defined and used by the developer of the plugin. The second section holds parameters used by the *PluginTag*. Currently, the following properties are available for the PluginTag section:

- *visibility*: holds a white space separated list of plugin names which are allowed to access this plugin. For external access (view) of the plugin, access must be granted to the *command* plugin name.
- *requires*: a list of plugin names separated by white space which is required for proper functionality of the plugin.
- *shared*: a Boolean value which can be *true* or *false* to indicate the that the plugin can accessed only exclusively.

A third section, which is optional, describes the transformation of input and ouput strings. This section is inspired by the define macro of C:

- A definition like "BEGIN>1" would translate the incoming word 'BEGIN' to '1'.
- A definition like "END<0" would translate the outgoing word '0' to 'END'.

[3] http://www.gnu.org/software/commoncpp

– A definition like "ROSES=2" would translate each incoming word 'ROSES' to '2' and the other way round.

The next step is to define the necessary functions that are available for the different views to the extension. The internal view, which is the same as the plugin interface, consists of function names of arbitrary signature whereas the external view has functions whose return values should be meaningful due to the conversion to a string. The *Common C++* library is used to define an active extension by using the threading classes of the library.

Plugin functionality In addition to the four standard functions *init*, *activate*, *deactivate*, and *emergencyHalt* the user defined functions as defined in the interface headers have to be implemented.

Compiling the plugin After defining the plugin internals, we are now able to generate optional wrapper classes and dispatcher functions for external access. This his handled by a python script which scans the external interface function declarations and generates the internal control structure for each class and the dispatcher. For the internal access the interface header files of the extensions are needed which are used by the plugin. These are usually located in the include directory which is defined by robrain. Alternatively the interface header files can be moved into the same directory that contains the plugin.

The plugin is now ready to compile. Afterwards, the generated shared object file is copied to the plugins directory of robrain. The configuration and interface header file (in case that the plugin should be internally accessible) are also copied to the according directories. A second copy with a different name of the same shared object can be copied into that directory while using different configuration file which correspond to the chosen name.

4.2 Runtime system

Dynamic plugin loading The dynamic loading of extensions is implemented using the *dlopen API* which is provided in C. For each shared object file in the plugins directory the file will be opened using the *dlopen* function. As shown in figure 3, for every shared object file (which is recognitized by a '*.so*' suffix) an instance is created if the file has the *GETPLUGIN* symbol which is detected by using the dlsym function. Due to name mangling of C++ the instantiation of the class has to wrapped around with *extern C*. It creates an instance of a class defined by the developer and returns the pointer to the system. In case of NULL pointer of *GETPLUGIN* this plugin will be dropped since every plugin has to include this symbol which is automatically done using the mentioned shell script *new_plugin.sh*.

Currently, new extensions are only available at the start of robrain. The loading of an extensions after the start sequence demands adding further functionality to the system. An aspect to take care of is that extension loading must

be done by carefully checking the dependencies of each plugins since an 'unresolved symbols' failure could terminate the overall system.

Plugin usage The two access methods of internal and external access require different handling.

The internal access is done by getting a pointer of type Plugin and using the dynamic_cast function of C++ to convert the object to the desired object class. Finally, one can access the extension functions like any other C++ object.

The external access requires the installation of the command plugin and the interactiveshell and/or the tcpsocket extension in order to set up calls like "call plugin function(...)".

command extension of type *Device* is responsible for parsing the incoming call encoded into a string object by calling the corresponding plugin function and returning the results in form of a string. Furthermore, a list of available plugins to outside to the system can be listed, which correspondents the visibility of the plugins to the *command* plugin. Therefore, the command plugin defines the link between the plugins which can be seen from external view and the higher level plugins which connects to the outside world. Each plugin which has to be seen from outside needs to be a visible to the command plugin.

interactiveshell extension of type *Unit* opens a command line interface where the functionality of the command plugin can be accessed.

tcpsocket extension of type *Unit* opens a socket port and forwards the incoming strings to the *command* plugin and vice versa.

4.3 Open Issues

Due to limitations of the programming language, we identified two issues that need to be solved in a future version of robrain:

Symbol loading Loading or removing an extension without checking dependencies could quit the system with an 'unresolved symbols' failure since the internal access between the extensions are implemented as a raw access using the pointer of the object after doing a *dynamic_cast* of the plugin to the appropriate extension class.

Data types Using the string stream operation to convert between any arbitrary type and the string type only those types are supported that can be handled by string streams. That means only C++ specific types and classes are allowed. In order to use non C++ types, the particular type can be wrapped with a class providing information how to deal with the *operator*<< and *operator*>> operators by defining them as friends of the class and define the internal handling of the values by overloading these operators. Figure 7 shows an example for the definition and the usage of a new type.

```
class Complex {
    friend ostream& operator<< (ostream& output,
                                const Complex& p);
...
private:
    int re;
    int im;
...
public:
...
}
ostream& operator<<(ostream& output,
                    const Complex& c) {
    output << "(" <<  c.re << ", " << c.im <<")";
    return output;
}
// example usage
Complex c;
cout << c;
```

Fig. 7. Example for definition and usage of new types

4.4 Available plugins

Several extensions have been already written for the architecture. Besides the core applications in order to use the robot, there are plugins which arise from multiple project and master theses.

Mocdevice We use the robot system Robertino, which comes with a simple C library for hardware access. The Mocdevice plugin provides hardware access and solves the need that only one instance of the library is used at a time. All the function calls are synchronized through a mutex. Based on this extension, higher level functionality like controlled motion, e.g. driving a given distance or angle will be performed by the *DriverUnit* extension. There is also an extension for measuring the battery status and for distance determination.

SNwExplorationUnit A sensor network plugin was created as part of a student's theses [5]. It provides necessary functionality to access a sensor network consisting of Mica2 motes. Currently, it collects data from sensor nodes and provides them to the system.

BM² In another project, a hardware module for smart power management (battery management and battery monitoring) was developed and adapted to the Robertino [3]. The software component was developed in form of a plugin for robrain which provides energy measurement and monitoring functionality.

5 Conclusions and Further Work

In conclusion it can be said that we were able to design, implement, and deploy a software architecture for use in autonomous mobile robot systems that fulfills all key requirements. The attempt of an easy extensible system for robot control resulted in a software architecture that addresses a wide application area. The system can easily be adapted to other robot platforms supporting a programming environment using C++. During the last year, we employed the developed robrain system as a basis for several project and master theses. We have discovered that several studies can be easily conducted due to the modular structure and the provided remote controllability. Additionally, the system was used in some courses. The students were rapidly able to use and extend the system depending on their needs.

More sophisticating methods like RPC-based access are planned because the external handling through a string limits the possible usage. There are also a work to integrate python as an easy to use language to provide task planning and extension development. Additionally, further student projects are going on to provide a GUI interface to the system and to use the vision hardware of the robot. Nearly every standalone software which is actually working (and implemented in C++) can easily be included into the system extending the existing functionality.

References

1. Dressler, F., Fuchs, G.: ROSES - robot assisted sensor networks. Poster, Dept. of Computer Sciences, University of Erlangen-Nuremberg (2004)
2. Dressler, F., Fuchs, G.: Energy-aware operation and task allocation of autonomous robots. In: 5th IEEE International Workshop on Robot Motion and Control (IEEE RoMoCo'05), Dymaczewo, Poland (2005) 163–168
3. Lucas, N., Codrea, C., Hirth, T., Gutierrez, J., Dressler, F.: RoBM: Measurement of battery capacity in mobile robot systems. In: GI/ITG KuVS Fachgesprch Energiebewute Systeme und Methoden, Erlangen, Germany (2005)
4. Dressler, F.: Sensor-based localization-assistance for mobile nodes. In: 4. GI/ITG KuVS Fachgesprch Drahtlose Sensornetze, Zurich, Switzerland (2005) 102–106
5. Frunzke, T.: Anbindung von Sensorknoten an mobile Roboter. Pre-master's thesis (studienarbeit), University of Erlangen-Nuremberg (2005)