MCA - An Expandable Modular Controller Architecture

K.-U. SCHOLL, J. ALBIEZ, B. GASSMANN
Forschungszentrum Informatik an der Universitaet Karlsruhe (FZI)
Interactive Diagnosis and Service Systems
Haid-und-Neu-Str. 10-14, 76131 Karlsruhe, Germany
albiez@fzi.de, http://www.fiz.de/ids/

Abstract

For effective development and implementation of different robot prototypes the use of transferable software components is a very important aspect. Only reusable components with standardized interfaces will result in extendable architectures. A reasonable approach to development and debugging is to make it possible to reuse these components in different environments (eg. Real-Time or Microcontroller) without having to change the code, This paper present a short overview of the modular software framework MCA which tries to meet this requirements.

1 Introduction

MCA is the acronym for **M**odular **C**ontroller **A**rchitecture. Its development started five years ago when a common software platform for all robots at our institute was required [1]. Over the years a first simple approach was enhanced more and more, until at the end of year 2000 the base code was completly rewritten. In April 2001 the first port of the user interface and part of the network and base systems to windows could be completed. This relative stable release was tagged as version two and released under GPL in May 2001 [10].
Efforts in software creation should be restricted to the development of new components and control methods not being implemented so far. If parts of a whole controller system are organized in small standardized modules, these modules can be reused in other projects. Communication between and synchronization of the different parts of the software has to be done by the controller system, so the project developers can focus their work on the methods necessary for controlling the robot.
Beside the reusability of controlling mechanisms for other projects the short training period of new students as well as the uniform software appearance on all levels has played an important role during the development of MCA. Therefore C++ as programming language has been chosen. To meet eventual real-time requirements Real Time Linux [6] has been used. RT-Linux combines real-time functionality with the advantages of a stable Unix system. In RT-Linux a simple real-time executive runs a non-real-time kernel as its lowest priority task.
This paper provides an overview of the inside functionality of MCA, the utilized tools and the implemented projects.

2 Architecture

MCA is constructed as a classical hierachical control strucutre. On the lowest level resides the hardware interface units, the more you ascend in the system hierachy the more abstract the units get. The user interface can be found on top of the system. In general there exist two dataflows inside MCA:

- **Sensor Data:** Sensor data is flowing upwards from the actual sensors through all modules requiring it up to the user interface. Modules in between this flow can modify it and/or create new sensor values.
- **Control Data:** Control data is flowing from the toplevel down to the actuators at the bottom.

The most basic unit of MCA is a *Module*. The modules can be put in a *Group*. Modules inside a group communicate with each other over free configurable *edges*. A group can be put inside another group or can be used as main loop inside a *part*. A part represents the main execution environment of MCA.
The next sections will discuss each component in detail.

2.1 Modules

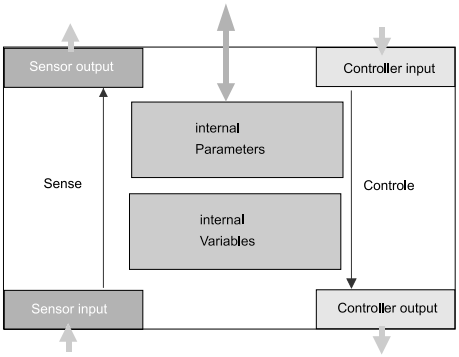

FIGURE 1: *The I/O of a MCA module*

A module is the basic building unit of a MCA controller system. Every module is a child of the class *BaseModule* which provides the management interface to the complete MCA system. It accepts sensor information from lower levels or the hardware level on a *sensor input*, processes these sensor values in a function *Sense()* and writes them back to the *sensor output*. Control information is coming from *control in*, is processed in a function *Controle()* and is written to *control out*. *Sense()* and *Controle()* can exchange information via local variables. The basic structure of such a module is shown in figure 1.
Control in, *Control out*, *Sensor in* and *Sensor out* are modelled as vectors of double. Each vector element owns a description, the whole vector has a flag for signalling the module that elements have been changed. This enables the code inside *Controle()* or *Sense()* only to be executed when new values have been arrived. The control and sensor data paths should only be used for data changing very frequently. Values used as control system parameters can be set over a second access method, called *Parameters*. Each module can possess a set of parameter values. Inside the module class this parameters can be accessed very easily via an array of floats. But as a specialfeature this array can be accessed from outside the module as well. Using the MCAadmin tool or something equal it is possible to access and change these parameters during run-time. Therefore it is not necessary to stop the system while fine-tuning. After this tuning the best value for each parameter can be compiled in as default value.
A typical example for this procedure can be a module calculating the kinematic problem of a robot. The position and the angles of the tool center point (TCO) are used as controller input. The module calculates the inverse kinematic problem in the *Controle()* function and writes the joint angles of the appropriate robot configuration to control out. The real values from the joints are put into sensor input, *Sense()* calculates the direct kinematic and dumps the TCP position and the tool orientation as sensor output. Maximum and minimum joint angles as well as the segment lengths of the robot can be saved as parameters.

2.2 Groups and Edges

The concept of MCA is to put basic functional blocks in modules. Since in most cases a control system will consist of more than one function block MCA provides a method for combining these blocks in *Module Groups* (figure 2).

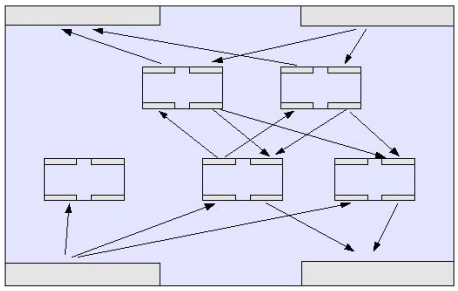

FIGURE 2: *An example for a MCA group*

Modules are added to a group at a specific *level*. The number of levels of a group has to be specified at creation time of the group. The class *ModuleGroup* is derived from the *BaseModule* class and therefore behaves like a module when accessed from the outside. A module group has a *Control in*, *Control out*, *Sensor in* and *Sensor out* vector and a description. It manages all modules added to it by references.
Datahandling between modules is done via *edges*. An edge connects a sensor output with a sensor input or a controller output with a controller input of two modules. An edge can pass the whole vector, only parts of it or an arbitrary permutation of the vector elements. This allows a quick and easy method two build a complex control structure inside a group. Since groups behave like modules they can easily be nested.
The *Controle()* and *Sense()* methods manage the data flow and the module execution inside a group. In case of *Sense()* this done in the following order:

1. Pass all edge data from sensor input of the group to all modules on the lowest level n of the group.

2. Call *Sense()* in all modules on the lowest level n.

3. Pass all sensor data between modules of the level n and $n-1$ in case they are connected by edges.

4. Call *Sense()* in all modules on the $n-1$th level.

5. Repeat step 3 and 4 for all $n = n - 1$ levels.

6. Update the sensor output of the group.

Data is only passed, if the flags in the output vectors are set, indicating new values have arrived. *Controle()* does the same for the controller input only starting on the first level and desecnding downwards.

2.3 Parts

A *Part* is the execution environment of MCA. Each part manages a module or a module group. The necessary execution environment, e.g. the process context on linux, is provided by the part. Parts can be connected to other parts running on different computers or in other environments (see next section). Data flow between these parts is also handeled by the part and passed to the managed module as sensor or control data. Each part does the following things periodically (for a given period in μs):

1. Pass all data from and to lower parts (if existent).

2. Call the *Sense()* function of the managed module.

3. Pass all data from and to highter parts.

4. Call the *Controle()* function of the managed module.

2.4 Real-Time and Network-Transparency

MCA offers complete network and realtime transparency. Viewed from the control system it does make no difference where a certain part is executed as long as every added part is told where the other part is running. It is up to the system designer to decide where and under which conditions each part should be runing (see figure 3.

This concept allows to develop a control strucutre in a safe Linux enviroment, fix all the bugs with normal tools like debuggers, and then move the whole system into a real-time task without modifications to the program code.

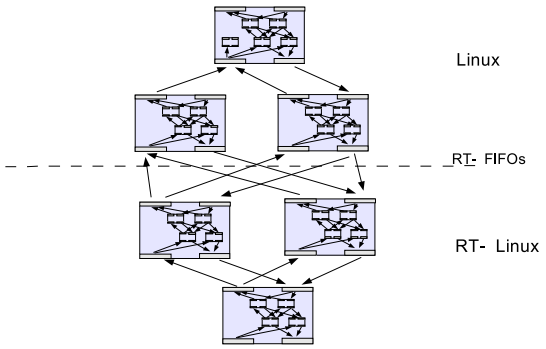

FIGURE 3: *An example for a system with three parts run in a RTLinux environment and three other parts in a normal Linux system*

For example all the time critical parts can run in a real time environment on a robot's onboard embeded system, while in the linux context of this system another part is started connecting itself to the first part using RT-Fifos. All things not time-critical or not RT safe can be done in this part. On a second computer connected via wireles LAN to the robot's PC runs a third part logging sensor values or controlling external periphery like cameras or a treadmill. This part connects to the system directly via TCP/IP. The controlling user interface and the MCAadmin now could run on a third computer (e.g. a laptop) used by the robots operator.

3 Tools

3.1 Admin

The Qt [5] based tool MCAadmin is the browsing utlity of the complete MCA structure (figure 4). It connects via TCP/IP to the first MCA part and can be used to display the complete structure in a tree form.

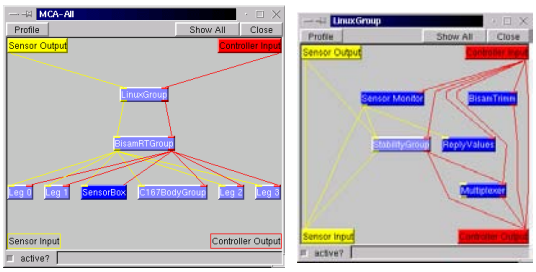

FIGURE 4: *MCAadmin browsing the strucutre of a control system*

MCAadmin also allows inspecting the parameters of a module, the connections of each edges, stopping a special module, inspecting and setting the controller/sensor inputs/outputs of a module, browsing recursively into module groups and printing the time percentage each module uses of the overall period of a part (see figure 5).

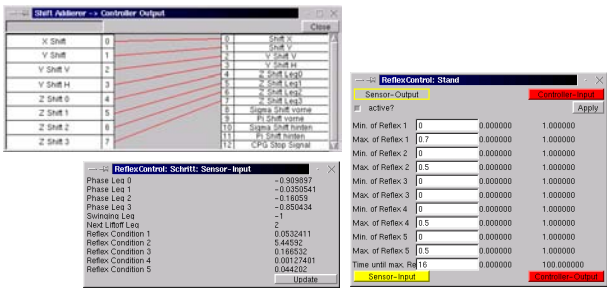

FIGURE 5: *MCAadmin inspectiing an edge and the parameters of a module*

3.2 User Interface

MCA provides a general fre configurable user interface called MCAGUI. MCAGUI connects via TCP/IP to the highest part of the control hierachy and can influence the controller inputs and sensor outputs of this part.

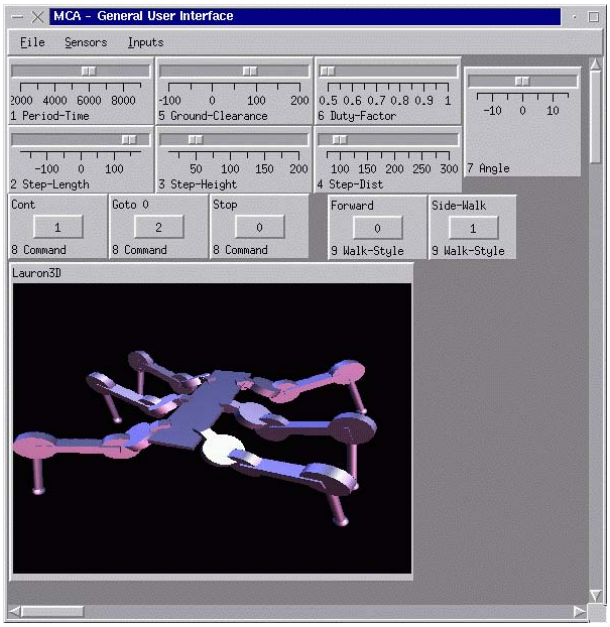

FIGURE 6: *A MCAGUI controlling the six-legged robot LAURON III*

The gui can be constructed dynamically by inserting widgets in the main canvas and connecting the inputs and outputs of the widgets to the controller and sensor intuts/outputs of the highest part. On Linux MCAGUI uses dynamical loading of widgets allowing to easily create custom widgets for special application. The basic builtin widgets include sliders, push buttons, LCD displays, radio button field, LEDs, switchboxes, and an interface to OpenInventor[9] for 3D representation of the robot and its environment.

4 Projects

Over the years MCA has been successfully used for all the robotic projects of our group (see figure 7, 8, 9, 10 and 11) .

FIGURE 7: *The six-legged walking machine LAURON III [4]*

FIGURE 8: *The four-legged walking machine BISAM [2]*

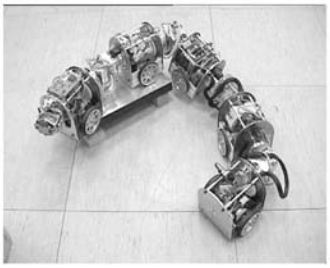

FIGURE 9: *The autonomous sewer inspection robot KAIRO [8]*

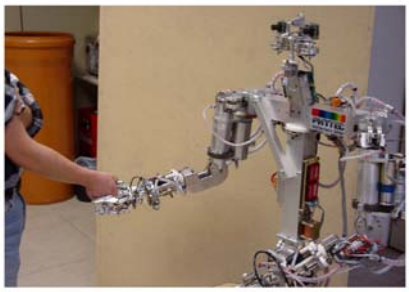

FIGURE 10: *The humanoid robot ARMAR [7]*

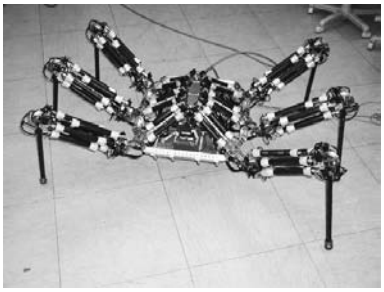

FIGURE 11: *The six-legged walking machine AirBug [3]*

5 Summary and Outlook

A scalable and modular controller architecture has been presented. With the help of this architecture it is possible to get new robot prototypes working very quickly and test them without having implementation work to do. The latter is only necessary if different mechanical aspects or new sensor components have to be considered.
The modular framework allows development and testing of new program code in Linux or Windows user space before the execution in a real-time environment or on micrcontrollers. With the network transparency it is possible to spread the controlling software over several computer thus allowing to use more time consuming non critical control methods without disturbing time critical real time processes. The user interface MCAGUI provides a flexibel method to control actors and visualize sensor information. The use of the browsing utility MCAadmin enables the fine tuning of system parameters and offers the posibility to display the acual control strucutre of the running system.
Future work will mainly focus on expanding the code base by more complex functions like machine learning, sensor information processing and behaviour control as well as on providing connection methods to different bus architectures like firewire. Another aspect is the handling of bulk data as produced by cameras and laserscanners between different parts. Also a lot effort is put in porting MCA to more architectures.
Always have a look at **http://mca2.sf.net/** for the newest updates on MCA.

References

[1] K.-U. Scholl, V. Kepplin, J. Albiez, R. Dillman, April 2000, *Develloping robot prototypes with an expandable modular controller architecture*, International Conference on Intelligent Autonomous Systems IAS 2000

[2] J. Albiez, T. Luksch, R. Dillmann, 2001, *Reactive Reflex based Posture Control for a Four-Legged Walking Machine*, Fourth international conference on Climbing and Walking Robots CLAWAR 2001

[3] Berns K., Albiez J., Kepplin V., Hillenbrand C., 2001, *Airbug - Insect-like Machine Actuated by Fluidic Muscle*, Fourth international conference on Climbing and Walking Robots CLAWAR 2001

[4] Gassmann B., Scholl K.-U., Berns K. *Behaviour Control of LAURON III for Walking in Unstructured Terrain*, Fourth international conference on Climbing and Walking Robots CLAWAR 2001

[5] TrollTech, *The Qt GUI Framework*, http://www.troltech.com

[6] FSM Labs, *Real Time Linux Extensions*, http://www.rtlinux.org

[7] T. Asfour, K. Berns, R. Dillmann, 1999, *The humanoid robot ARMAR*, The second Internaional Symposium on Humanoid Robots 1999

[8] K.-U. Scholl, V. Kepplin, K. Berns, R. Dillmann, 1999, *An articulated service robot for sewer inspection tasks* IEEE/RSJ International Conference on Intelligent Robots and Systems

[9] Josie Wernecke, Open Inventor Architecture Group, 1994, *The Inventor Mentor*, Addison-Wesley, ISBN 0-201-62495-8

[10] Modular Controller Architecture, http://mca2.sf.net