FP6-004381-MACS

MACS

Multi-sensory Autonomous Cognitive Systems Interacting with Dynamic Environments for Perceiving and Using Affordances

Instrument: Specifically Targeted Research Project (STReP)

Thematic Priority: 2.3.2.4 Cognitive Systems

D1.3.1 Integrated Implementation of Reference Control System

Due date of deliverable: May 31, 2006
Actual submission date: July 13, 2006

Start date of project: September 1, 2004 Duration: 36 months

Fraunhofer Institut für Intelligente Analyse- und Informationssysteme (FhG/IAIS)

Revision: Version 1

Project co-funded by the European Commission within the Sixth Framework Programme (2002–2006)		
Dissemination Level		
PU	Public	**X**
PP	Restricted to other programme participants (including the Commission Services)	
RE	Restricted to a group specified by the consortium (including the Commission Services)	
CO	Confidential, only for members of the consortium (including the Commission Services)	

EU Project MACS

Deliverable 1.3.1

Integrated Implementation of Reference Control System

Rainer Worst, Hartmut Surmann, Martin Hülse

Number:	**MACS/1/1.2**
WP:	*1.1*
Status:	*Version 1*
Created at:	*July 8, 2006*
Revised at:	*July 13, 2006*

FhG/AIS	Fraunhofer Institut für Intelligente Analyse- und Informationssysteme, Sankt Augustin, D
JR_DIB	Joanneum Research Graz, A
LiU-IDA	Linköpings Universitet, Linköping, S
METU-KOVAN	Middle East Technical University, Ankara, T
OFAI	Österreichische Studiengesellschaft für Kybernetik, Vienna, A

This research was partly funded by the European Commission's 6th Framework Programme IST Project MACS under contract/grant number FP6-004381. The Commission's support is gratefully acknowledged.

Corresponding author's address:

Rainer Worst
Fraunhofer Institut für Intelligente
Analyse- und Informationssysteme
Schloß Birlinghoven
D-53754 Sankt Augustin, Germany

Fraunhofer Institut für Intelligente Analyse- und Informationssysteme Schloss Birlinghoven D-53754 Sankt Augustin Germany	Tel.: +49 (0) 2241 14-2683 (Co-ordinator) **Contact:** Dr.-Ing. Erich Rome
Joanneum Research Institute of Digital Image Processing Computational Perception (CAPE) Steyrergasse 9 A-8010 Graz Austria	Tel.: +43 (0) 316 876-1769 **Contact:** Dr. Lucas Paletta
Linköpings Universitet Dept. of Computer and Info. Science Linköping 581 83 Sweden	Tel.: +46 13 24 26 28 **Contact:** Prof. Dr. Patrick Doherty
Middle East Technical University Dept. of Computer Engineering Inonu Bulvari TR-06531 Ankara Turkey	Tel.: +90 312 210 5539 **Contact:** Prof. Dr. Erol Şahin
Österreichische Studiengesellschaft für Kybernetik (ÖSGK) Freyung 6 A-1010 Vienna Austria	Tel.: +43 1 5336112 0 **Contact:** Prof. Dr. Georg Dorffner

Contents

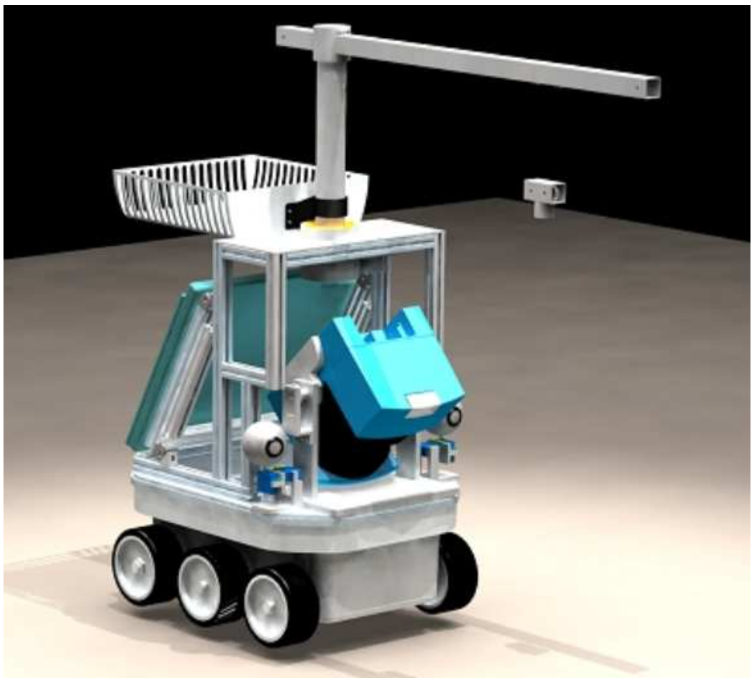


Figure 1: MACS edition of KURT3D (CAD model)

1 Introduction

In this document, we describe the first iteration of the Reference Control System (RCS) implementation. Up to now the RCS basically provides for all the partners the essential functionality that the robot KURT3D is delivering and as it is relevant for the MACS project. The RCS uses CORBA TAO as Middleware and implements the IDL interfaces of the KURT3D system as introduced in [4]. The CORBA based implementation of the interfaces at this level allows us the integration of the real KURT3D hardware as well as its physical simulation MACSim [2] into the framework of the MACS project in an easy-to-use way.

We start with a documentation of the robotic platform KURT3D, which is the base for the reference control system in the MACS project. The platform's hardware consists of power supply, actuators, sensors, microcontrollers, and the on-board PC system. The PC is connected via CAN bus to an Infineon C167CR microcontroller inside the platform and to an Infineon C164 microcontroller attached to the crane. Both, the entire hardware and the firmware for the microcontrollers is described.

In the following, we give an overview of the current implementation of the RCS. This includes its logical organization as well as the description of the corresponding source code in the MACS repository (for details about general setup of the MACS CVS repository see [3]).

A simple example is also described as a demonstration of method in order to provide all project partners with the essential information for their first usage of the RCS.

The document concludes with a brief overview of future steps that will lead to the integration of basic behaviors of the KURT3D system, like explore and approach behaviors.

2 Robot Hardware

This section describes the hardware of the wheeled robot KURT3D (cf. Fig. 1), which has been chosen to be the standard platform in the MACS project. The description shall provide an overview on all the components of the system without going in deep detail regarding these components. It is recommended to study the manuals or data sheets of the parts mentioned to get the entire information if necessary.

2.1 Power Supply

The supplied battery pack for the platform consists of 20 NiMH cells with a capacity of 4,500 mAh. So we get a nominal voltage of 24 V. This voltage is transformed on the mainboard by two DC/DC converters to 12 V resp. 5 V. Therefore, the mainboard offers three different voltages, which are used typically as follows:

- 24 V: motors, sonar sensors
- 12 V: CCD camera
- 5 V: microcontroller, tilt sensors, shaft encoders, infrared sensors

A second battery pack of the same size is used for the power supply of the 3D laser range finder. Finally, there is a third battery pack consisting of 4 NiMH cells (nominal voltage 4.8 V) with a capacity of 5,000 mAh. This pack is used for the power supply of the servos, which move the laser range finder and the cameras, as well as for the corresponding interface board between RS232 and servo control. Additionally, one has of course also to take care of the notebook PC's battery, without any specific requirements regarding KURT3D.

2.2 Actuators

2.2.1 Wheel Drives

The robot has three wheels on each side, which are connected by a tooth-belt. A single Maxon DC motor drives each side. The motor can be operated between 2 and 42 V with a maximum power consumption of 90 W. A planetary gear with a reduction of 35:1 is mounted at the motor, thus allowing a torque of 1 Nm. Furthermore, a shaft encoder is mounted at the motor, which provides a periodical, digital signal and allows to measure the number of revolutions. The encoder delivers 500 ticks per motor revolution, i.e. 17,500 ticks per wheel revolution. The output signals are directly connected to a timer of the microcontroller, which is used as a counter in incremental interface mode in order to distinguish between the two possible directions.

The current consumption of each motor lies between 0.3 and 1.5 A. These currents are delivered by a motor controller board, which is plugged in a socket on the mainboard. The motor voltage is controlled by pulse width modulation. The necessary pwm signals are generated by the microcontroller and sent to an IC of type LMD18200, which is the main part of the motor controller board. According to a specific logic, the IC produces the demanded output voltage depending on the settings of its pwm, brake, and direction pins. The actual motor current can be measured using Pin 8 of the IC, where a proportional current (377 mA/A) is provided.

2.2.2 Crane

The crane is not a standard component of the KURT3D system, but is a custom design for the MACS project. It is needed to enhance the manipulation capabilities of the robot, which is obviously essential in the context of affordances.

2.3 Sensors

2.3.1 Sonar Sensors

Some KURT3D systems use sonar sensors of the type Baumer-Electric UNDK 30U6103 as distance sensors. These sensors provide a voltage that is proportional to the distance of the obstacle, which reflects the echo. The dihedral angle is $10°$ and the usable measurement range is 10 - 70 cm. The output voltage is connected to a potential divider that reduces the voltage to an appropriate value for an analog/digital converter (ADC) pin of the microcontroller.

2.3.2 Infrared Sensors

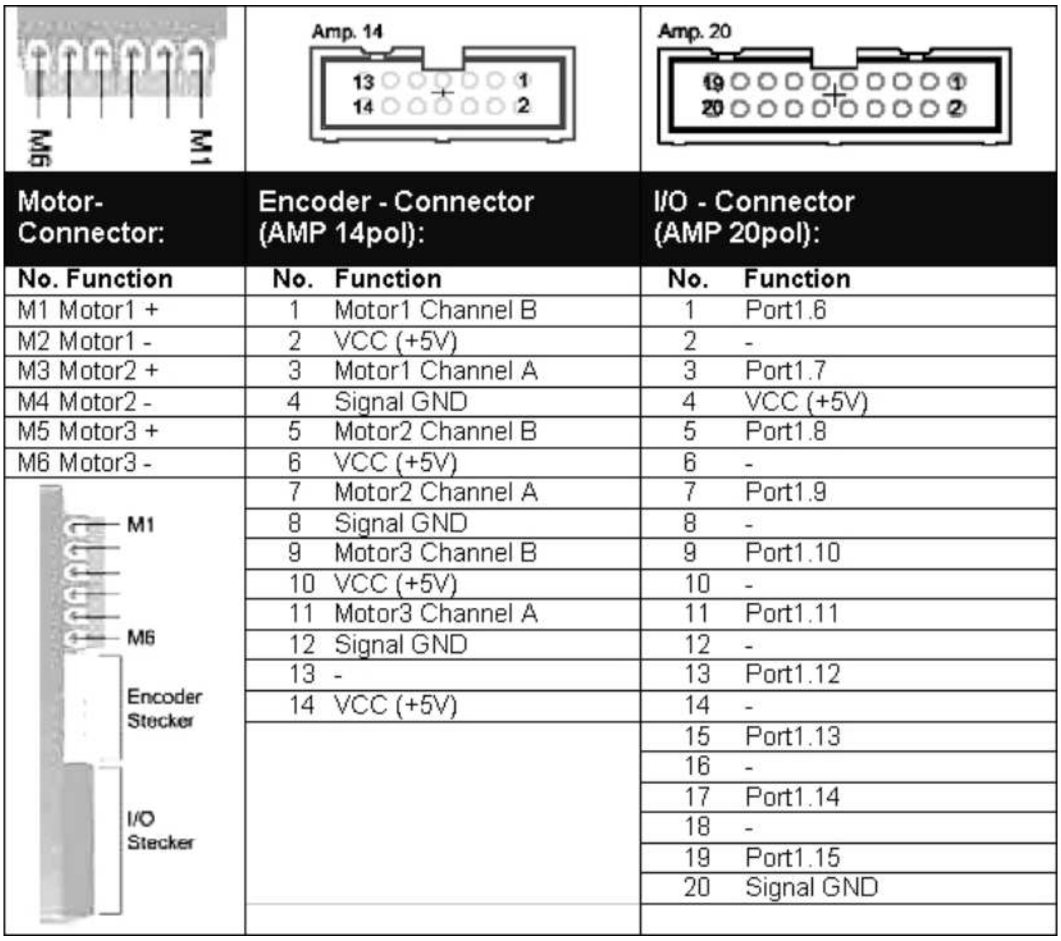
Most KURT3D systems have Sharp GP2D12 infrared sensors around their body to detect obstacles before they are touched. They provide a voltage that is in inverse proportion to the obstacle's distance. The output voltage (0 - 2.5 V) is connected directly to an analog/digital converter (ADC) pin of the microcontroller. The detecting distance is 10 - 80 cm.

2.3.3 3D Laser Scanner

- 2D laser scanner SICK LMS-200 including cables for KURT2 robot
- Suspension unit for the SICK laser scanner LMS-200
- Volz servo motor including safety clutch
- Serial microcontroller board to control up to 8 servo motors incl. cable
- SeaLINK RS 422 - USB connector or RS422 PCMCIA card to connect the scanner to a PC/laptop

2.3.4 Pan/Tilt Camera System

- 2 Volz servos
- Webcam Logitech Quickcam Pro 400
- Suspension unit for the servos and camera
- Serial microcontroller board to control up to 8 servo motors incl. cable


M6
M1
Amp. 14
13
14
1
2
Amp. 20
19
20
1
2
M1
M6
Encoder Stecker
I/O Stecker

Motor-Connector:		Encoder - Connector (AMP 14pol):		I/O - Connector (AMP 20pol):	
No.	Function	No.	Function	No.	Function
M1	Motor1 +	1	Motor1 Channel B	1	Port1.6
M2	Motor1 -	2	VCC (+5V)	2	-
M3	Motor2 +	3	Motor1 Channel A	3	Port1.7
M4	Motor2 -	4	Signal GND	4	VCC (+5V)
M5	Motor3 +	5	Motor2 Channel B	5	Port1.8
M6	Motor3 -	6	VCC (+5V)	6	-
		7	Motor2 Channel A	7	Port1.9
		8	Signal GND	8	-
		9	Motor3 Channel B	9	Port1.10
		10	VCC (+5V)	10	-
		11	Motor3 Channel A	11	Port1.11
		12	Signal GND	12	-
		13	-	13	Port1.12
		14	VCC (+5V)	14	-
				15	Port1.13
				16	-
				17	Port1.14
				18	-
				19	Port1.15
				20	Signal GND

Figure 2: Connecting the TMC200

2.4 Microcontrollers

The mainboard of KURT3D has two sockets for so-called miniModules-167 by Phytec. The first socket must always be loaded, but the second one is optional, e.g. to provide more ADC channels. Such a miniModule-167 contains essentially an Infineon C167CR microcontroller plus external RAM and flash memory. The C167CR has an on-chip CAN interface, which is used to connect the two microcontrollers. Further details about the C167CR can be found in the user's manual available from the Infineon website.

The crane's motors are controlled by a separate TMC200 board, which is able to control up to 3 motors with a power of 200 W. The pin out of the connectors at the motor side of the TMC200 is shown in Fig. 2.

The mapping of the crane's sensors and motors with the ports of the TMC200 board is shown in the following table:

Function	Type	Signal	Port TMC200
Encoder Motor 1 A	CAPCOM	pulse	EC-Pin_3
Encoder Motor 1 B	CAPCOM	pulse	EC-Pin_1
Motor 1 VCC	POWER 5V	-	EC-Pin_2
Motor 1 GND	POWER GND	-	EC-Pin_4
Encoder Motor 2 A	CAPCOM	pulse	EC-Pin_7
Encoder Motor 2 B	CAPCOM	pulse	EC-Pin_5
Motor 2 VCC	POWER 5V	-	EC-Pin_6
Motor 2 GND	POWER GND	-	EC-Pin_8
Encoder Motor 3 A	CAPCOM	pulse	EC-Pin_11
Encoder Motor 3 B	CAPCOM	pulse	EC-Pin_9
Motor 3 VCC	POWER 5V	-	EC-Pin_10
Motor 3 GND	POWER GND	-	EC-Pin_12
Limit switch Axis 2 front	Input	high/low	IOC-Pin_5
Limit switch Axis 2 back	Input	high/low	IOC-Pin_7
Limit switch Axis 3 top	Input	high/low	IOC-Pin_9
Magnet on/off	DA / output	-	IOC-Pin_1
Position Axis 1	analog input	frequency	IOC-Pin_13
Weight	analog input	frequency	IOC-Pin_15

2.5 PC

KURT3D is controlled at the application level by an arbitrary notebook PC that is connected to the on-board CAN bus, which is used for the communication between the PC and the microcontroller(s). Since CAN is not a standard interface for PC systems, one has to insert a PC Card or use a USB to CAN adapter. With KURT3D, CAN specific interface cards by Softing or MicroControl are supported, but in the MACS project an USB to CAN adapter by Systec is preferred. The notebook PC should also provide a PC Card slot for Wireless LAN. A serial interface is needed to connect the interface board for the servos. Finally, three USB ports (maybe an USB hub is necessary) are used to connect the two cameras and the RS422 adaptor, which provides the Data from the Laser scanner.

3 Robot Firmware

The firmware for the KURT3D robot is available as open source code from the site `http://www.ais.fraunhofer.de/KURT2/resources.htm` .

3.1 Installation Instructions

The file k2167vvv.zip contains a complete distribution of the firmware for IAIS's robot platform KURT3D, where vvv indicates the version. A PC running Windows NT 4.0, 2000, or XP is required. You can either use the built-in PC system or any other PC. If you are going to modify the firmware you may need the Keil C166 Developer's Kit.

The firmware distribution consists of the directory `\kurt2167` . Unpack this directory to an arbitrary location, let's say `YOURPATH\kurt2167` . You will need the files of this directory for two different purposes:

- To actually write the firmware into KURT2's flash memory
- (optionally) To study the source code and modify it according to your needs

To write the firmware into the flash memory, do the following:

1. Unmount the cover plate of the robot
2. Connect the serial ports of the C167 and of the PC
3. Start Windows and log on
4. Open a MS-DOS shell (Start → Programs → Command Prompt)
5. Change directory to `YOURPATH\kurt2167\flash` (as created above)
6. Set C167 to bootstrap mode by pressing both boot and reset button, then releasing re-set button first
7. Execute programm "flasht 2" (assuming you are using COM 2)
8. Choose command "7" and load file "kurt2.h86" to flash memory
9. Be sure that the red LED1 is blinking with 1 Hz after Software-Reset
10. Perform the obvious finishing and re-mounting activities

Important Note: If you are using Windows 2000 or XP it is strongly recommended to install the Phytec FlashTools and to use them instead of the program "flasht" as described above. In the directory `YOUR-PATH\kurt2167\wintools\phytec` you can find this software ready for setup. Use the tools as an usual Windows application and proceed in this way:

1. Select target hardware: MINIMODUL/MINIMODUL-167/MODE1
2. Press connect button
3. Select interface (e.g. COM2) and baud rate (e.g. 57600)
4. Press OK button
5. Set C167 to bootstrap mode by pressing both boot and reset button, then releasing re-set button first
6. Select all sectors and press "erase sectors" button
7. Open the file `YOURPATH\kurt2167\flash\kurt2.h86` and press the download button
8. Press and release the C167's reset button

If your KURT2 system is equipped with a second microcontroller you may write the firmware into its flash memory in the same way, except of using the file kurt22.h86 instead of kurt2.h86 and of course connecting the PC to the serial port of the second MC.

To modify the source code you need an installation of the Keil C166 Software Developers Kit. Then simply double-click at the project file kurt2.uv2 (resp. kurt22.uv2) within the directory `YOURPATH\kurt2167` to open the project. The modifiable code consists of:

`kurt2mc.h`	`header file`
`imu.h`	`header file for IMU related functions`
`kurt2mc.c`	`main control loop (main MC)`
`kurt22mc.c`	`main control loop (optional 2nd MC)`
`behavior.c`	`default behavior without PC control`
`kurt2can.c`	`CAN related functions`
`imu.c`	`IMU related functions`

After building a new executable you can load it to RAM using the built-in monitor. Attention: be sure to use the correct monitor settings (e.g. COM 2 and 57600 baud). When you have finished debugging, open a Command Prompt Window, change directory to the corresponding subdirectory `YOURPATH\kurt2167\flash` and execute ”make.bat”, thus creating new files ”kurt2.h86” and ”kurt22.h86”.

3.2 Control Loop

A continuous loop with a frequency of 100 Hz is executed by the standard firmware. That means, every 10 msec the following actions are achieved by the main C167 using the CAN bus:

1. Sense motor encoder values (speed)
2. Receive a motor control command sent by the PC
3. Execute the received motor control command
4. Transmit info message
5. Convert and transmit analog sensor values
6. Transmit motor encoder values (speed)
7. Transmit bumper and remote control values
8. Update and transmit odometry related values
9. Transmit gyroscope data as estimated orientation

10. Query and transmit compass values
11. Query and transmit tilt sensor and temperature values (only every 2 seconds)

The optional second C167 executes a loop with the same frequency, regarding the sensors that are attached to it:

1. Transmit info message
2. Convert and transmit analog sensor values

Hence, a control program executed by the PC system may receive all appropriate sensor values and supply motor commands according to desired behaviors without taking care of the very low level details of microcontroller programming.

If the microcontroller eventually receives no new messages from the PC, e.g. because of a disconnected CAN bus, a default behavior is executed (e.g. all motors are stopped for safety reasons).

3.3 CAN Interface

There are three types of CAN messages used within the system:

1. Control messages
2. Info messages
3. Sensor messages

Control messages always have CAN Id. 1 resp. 17 and are discriminated by the first two bytes of the data frame defining the control mode. Control messages have to be sent by the PC system, which controls the robot. Info messages have CAN Id. 4 resp. 20. They are continuously sent by the microcontroller and contain information regarding hardware identification, firmware version, and the current value of the firmware's loop count. Sensor messages have a CAN Id. > 4 resp. 21. They are continuously sent by the microcontroller. CAN Id.s > 15 are used by the optional second microcontroller. The complete specification of all CAN messsages used by KURT3D can be found in the appendix A.

4 Application Software

The functions of the PC software, which controls the KURT3D robot, is documented in its current version as a html-file, generated by doxygen. It is accessible on the page http://www.macs-eu.org/software.html for review purposes. This documentation includes all public functions that are implemented by the C++ classes for KURT3D. Only a subset of these functions is used by the higher level software modules like Perception, Learning, and Deliberation. They use the robot as a CORBA server with the API that is specified in [4].

The same API is implemented for the simulator, which is described in [2]. The idea is that the control software is implemented as a CORBA client, which can connect either to a real robot or to the simulator. The next section shows how to do this.

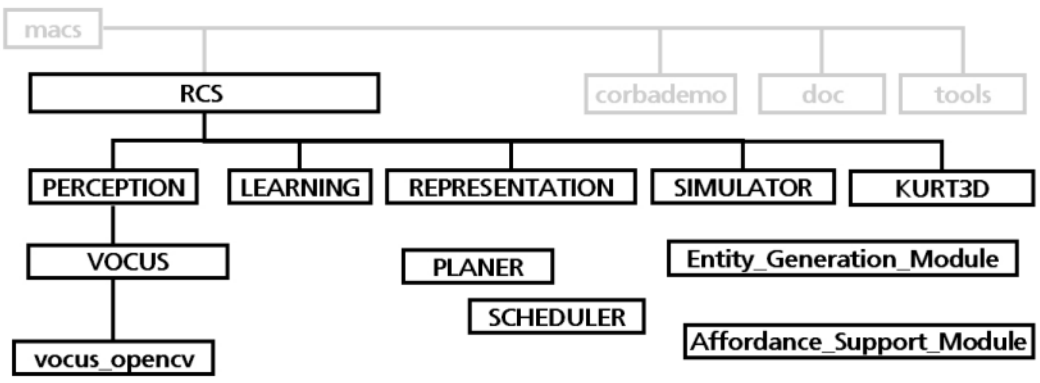


Figure 3: Structure of RCS in the repository

The software modules of the RCS of course reflect the affordance-based architecture that is to be developed in the MACS project. Although there is a rapid progress in defining the functionalities of the system, the architecture is not yet specified on a level that would allow to start with the implementation of all modules. This will be an incremental process and implies that the software will be changed accordingly. Fig. 3 shows the structure of the software in the cvs repository from the current perspective.

5 Integration of the Reference Control System

5.1 Setting up CORBA Servers and Clients

All project partners have access to the implementation of the IDL KURT3D interfaces [4] due to the CVS repository of the MACS project [3]. If one checks out the project `macs` in a certain directory, let say the home directory `$HOME`, the RCS implementation can be found in the directory:

```
$HOME/macs/AIS/RCS/
```

This folder includes the following three subdirectories:

```
CorbaClient
CorbaServerKURT3D
CorbaServerMacsim
```

As the names of the subdirectories are indicating they contain the server and client applications according to the corresponding KURT3D IDL interfaces. There are two directories for the server implementation since we have two types the KURT3D systems, namely the simulation (MACSim) as well as a real robot system. A client application is connected either to the real hardware or the simulation, depending which type of server is running.

IAIS and METU-KOVAN are responsible for the implementation of the two servers. The corresponding source code should not be touched by the other partners. Client applications do not have to take into account how a server is actually implemented.

Once the CORBA TAO environment has been successfully installed (see [4] for this procedure), the compilation and the start of the CORBA server for the real KURT3D platform is straight forward. After a change to the directory:

```
$HOME/macs/AIS/RCS/CorbaServerKURT3D
```

The server called `serverKURT3Dall` can be created by executing the command `make`.

But before this server can be started one has to start the CORBA name service first. Hence, the following two commands must be executed one after one:

```
>
>$TAO_ROOT/orbsvcs/Naming_Service/Naming_Service
                          -ORBEndPoint iiop://localhost:23456
>
>serverKURT3Dall
     -ORBInitRef NameService=corbaloc:iiop:localhost:23456/NameService
>
```

For details about these commands please see [4].

If a server is running an arbitrary client can be started in order to receive data from the sensors as well as to send data to the actuators of the KURT3D server, no matter if the server is actually based on the simulation or on the real robot hardware. A possible client application can be found in the directory:

```
$HOME/macs/AIS/RCS/CorbaClient
```

After successful compilation of the client by executing the command `make` in this directory, the client can be started in the same directory by the command:

```
clientKURT3D
   -ORBInitRef NameService=corbaloc:iiop:localhost:23456/NameService
```

Currently our first example of a CORBA client, which uses the KURT3D IDL interface implementation, is basically implemented by the three files:

```
client.cpp
RcsClient.cpp
RcsClient.h
```

The file `client.cpp` contains the main routine. In this function an instance of the class `RcsClient` is used to implement a client application. In our examples one can see that a `RcsClient` has at least five public member functions. These five function are sequentially executed:

```
int main(int argc, char * argv[])
{
    RcsClient client = RcsClient();

    client.openConnectionToServer(argc,argv);
    client.initComponents();
    client.run();
    client.stopComponents();
    client.closeConnectionToServer();
...
}
```

The first and the last method call of the `RcsClient client` provide the opening and the closing of the connection to the CORBA server.

Two additional methods can be used to bring the system in a defined state. This becomes obviously essential at two points in our client application: after a connection is established and before an application is actually starting (`initComponents()`) as well as before the connection to the server is closed (`stopComponents()`).

The method `run()` implements the loop that provides the computation of received data and the transmission of the data between server and client. The implementation of this method shows that an additional function `singleStep(...)` is executed as long as the return value of this function is `true`.

```
void RcsClient::run(){
    bool   goOn = true;
...
    do{
        goOn = singleStep(...);
    } while(goOn);
    return;
}
```

Therefore the main functionality of the client is determined by the private member function `singleStep(...)`.

The best way to understand this framework is to implement a concrete example. In the following chapter, we introduce such a concrete example in order to provide simple templates for all colleagues, who are interested to write their own client applications.

5.2 Example for the usage of the CORBA based RCS

As a demonstration of method for the implementation of a client using the KURT3D IDL interface, we introduce a task that drives a KURT3D pan-tilt-camera in such a way that the focus of attention (FOA) is in the center of the current image. The FOA is computed by the VOCUS software introduced in [1].

According to our software environment as introduced above the connection to the CORBA server is already implemented by the two functions `openConnectionToServer()` and `closeConnectionToServer()`. Without any changes they can directly be used for other client applications. Hence, we only have to describe the essential functionality of

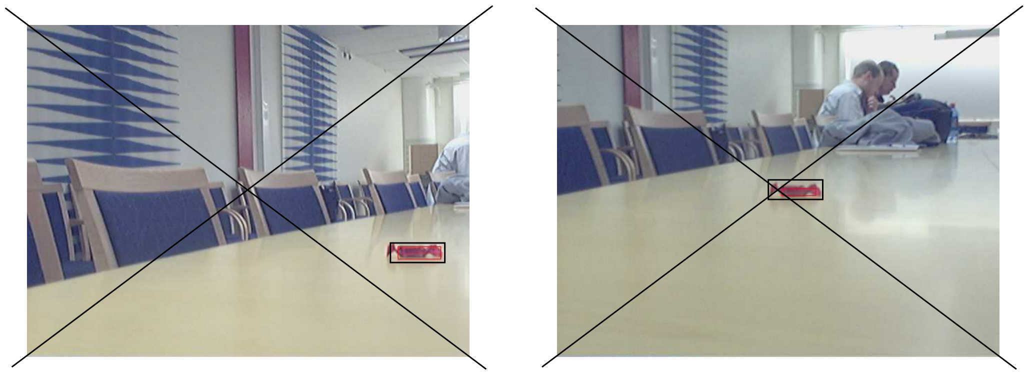


Figure 4: The images of a pan-tilt camera indicate view of the camera before (left) and after (right) the orientation to the current focus of attention. The black crossing lines are added in order to indicate the center of the image.

the private member function `singleStep(...)`, since this is the only place which has to be modified, beside the obvious initialization and clean-up procedures performed in the functions `initComponents()` and `stopComponents()`.

First it is important to notice that the interfaces of the KURT3D system are the arguments of the function `singleStep`. Therefore function calls as defined in the corresponding IDL interface specifications can be implemented by using the name of these arguments.

```
bool RcsClient::singleStep(Macs::IBasicControl_var basicControl,
                           Macs::ICamera_var cameras,
                           Macs::ILaserScanner_var laserScanner){
...
        Macs::ICamera::RgbImageArray *rgbImage;
...
        // get image data
        success = cameras->getRgbImage(Macs::ICamera::LEFT, rgbImage);
...
        // change pan and tilt posistion
        cameras->changeCameraPanTilt(Macs::ICamera::LEFT,
                                     (signX*changeSteps),
...                                  (signY*changeSteps));
}
```

In this example one can see how an image is actually requested from the left camera and how the pan and tilt positions are changed by the values of the second and third arguments. Just to remember, the two functions and the data type for the image data etc. are defined in the interfaces `ICamera` of the corresponding IDL file (see [4]):

```
module Macs
{
...
 interface ICamera
 {
  enum Side {LEFT, RIGHT};
...
  typedef sequence<octet> RgbImageArray;
...
  typedef long Pan;
  typedef long Tilt;
...
  boolean changeCameraPanTilt(in Side s, in Pan panOffset, in Tilt tiltOffset);
...
  boolean getRgbImage(in Side s,out RgbImageArray data);
}
...
}
```

Coming back to our example and without going further into the details, the following procedure drives the camera step by step in the direction of the FOA until the the FOA is in the center of the current image:

1. set camera to default position
2. compute the current FOA (using bottom-up mode)
3. orient camera towards this FOA (top-down mode) ...
4. until the given FOA is in the center of the image

The complete source of this procedure is given in the appendix B and in Fig. 4 an example of this application is shown.

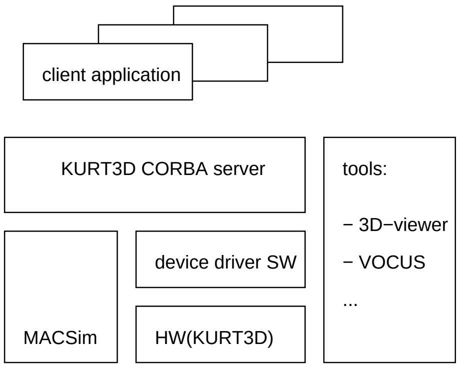
6 Summary and Future Work

The current state of the RCS via CORBA TAO makes it possible to write one application that is able to deal with the real KURT3D robot systems as well as its physical simulation MACSim (Fig. 5(a)).

Up to now the implementation of both servers is still in progress. But the implemented interfaces can already be used by the project partners. The functionality of pan-tilt camera is ready for use and can be applied to provide partners with real sensor data independently of other components, e.g. the crane.

The next steps towards an extension of the RCS is the definition and implementation of IDL interfaces which enable other applications to trigger specific basic behaviors, while sensor and actuator data can be received. Basic behaviors means that the triggered processes let the robot run autonomously.

In order to maintain the current integration of the real robot and the simulation, the needed basic behaviors have exclusively to use the KURT3D interfaces as specified in [4]. As a consequence an interface for specifying the triggering of these behaviors will be defined on a higher level (indicated in Fig. 5(b)).

(a)


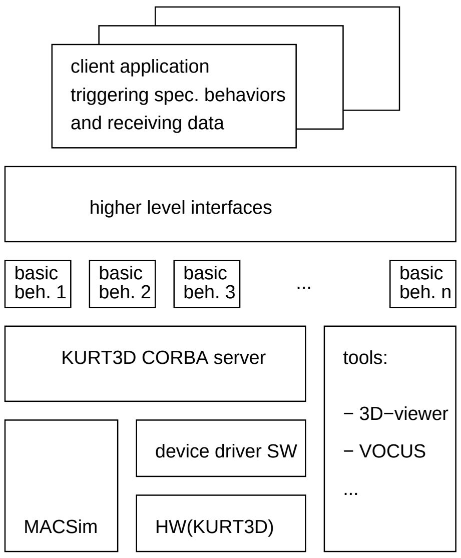
(b)


Figure 5: (a) The current state of the RCS implementation based on the KURT3D interface specification, which integrates the real robot as well as the its physical simulation. (b) Scheme indicating the implementation of the basic behaviors used by other application for triggering of such behaviors.

References

[1] Frintrop, S., Hülse, M., Rome, E., and Paletta, L. Saliency Detection with Visual Attention. Deliverable MACS/3/1.3 v1, Fraunhofer Institut AIS, Sankt Augustin, Germany, 2005.

[2] Ugur, E., Dogar, M. R., Soysal, O., Cakmak, M., and Sahin, E. MACSim: Physics-based Simulation of the KURT3D Robot Platform for Studying Affordances. Deliverable MACS/1/2.1, Fraunhofer Institut AIS, Sankt Augustin, Germany, 2005.

[3] Worst, R., Breithaupt, R., Hoffmann, C., and Surmann, H. Implementation of the Software Development Environment. Deliverable MACS/1/3.1, Fraunhofer Institut AIS, Sankt Augustin, Germany, 2005.

[4] Worst, R., Hoffmann, C., Wingmann, B., Cakmak, M., and Hülse, M. Specification of module interfaces. Deliverable MACS/1/1.2 v2, Fraunhofer Institut AIS, Sankt Augustin, Germany, 2005.

A Specification of KURT3D CAN messages

CAN Id. 1: The motors or other devices are controlled by a single CAN message, depending on the selected control mode.

0x0000 RAW control mode: The two motors are controlled by three parameters: direction of rotation, brake (switch motor on or off), and pulse width (similar to voltage). Direction is controlled by one bit, where '0' means "forward" and '1' means "back". Brake is controlled by another bit, where '0' means "brake off" (i.e. actually drive) and '1' means "brake on". Pulse width for left and right motor may be set separately to a value between 0 and 1024, where the meaning of 0 is "full power", 256 is corresponding to a duty cycle of 75 %, 512 is corresponding to a duty cycle of 50 %, and the meaning of 1023 is "no power at all". The intermediate values are treated in an analogous manner. This definition is in direct accordance with the operation of the C167's pulse width modulation module as described in chapter 15 of the C167 user's manual.

0x0001 SPEED control mode: The speed for left and right motor may be set separately to a value between -100 and +100, where the meaning of -100 is maximum speed back, 0 means no speed at all, and the meaning of +100 is maximum speed forward. The intermediate values are treated in an analogous manner. The desired speed will be maintained by the microcontroller.

0xFF00 ACTUATOR_ON control mode: This message causes the microcontroller to switch on an actuator, which is connected to its Pin 2.9.

0xFF01 ACTUATOR_OFF control mode: This message causes the microcontroller to switch off an actuator, which is connected to its Pin 2.9.

0xFF02 CALIBRATE_GYRO control mode: This message causes the microcontroller to recalibrate the (optional) gyroscope.

0xFF03 RESET_GYRO control mode: This message causes the microcontroller to reset the (optional) gyroscope.

0xFF04 RESET_SSC control mode: This message causes the microcontroller to reset the SSC interface (SPI), which is used for communication with the optional gyroscope module.

0xFF05 UPDATE_ANGLE control mode: This message updates the orientation component of the odometry with a new angle.

0xFFFF MC_RESET control mode: This message causes a software reset of the microcontroller.

CAN Id 4: The hardware identification represented by an unique number and the firmware version are transmitted in 2 groups of 2 bytes. The current value of the firmware's loop count is transmitted in 4 bytes.

CAN Id 5: The current values of sonar sensors 0 - 3 (attached to channels 0 - 3 of the C167's analog/digital converter, resolution: 10 bit) are transmitted in 4 groups of 2 bytes.

CAN Id 6: The current values of sonar sensors 4 - 7 (attached to channels 4 - 7 of the C167's analog/digital converter, resolution: 10 bit) are transmitted in 4 groups of 2 bytes.

CAN Id 7: The current values of sonar sensors 8 - 9 (attached to channels 8 - 9 of the C167's analog/digital converter, resolution: 10 bit) are transmitted in 2 groups of 2 bytes, followed by the values of adc channel 10 and adc channel 11 transmitted in 2 groups of 2 bytes.

CAN Id 8: The current values of the left and right motor's current sensors (attached to channels 12 - 13 of the C167's analog/digital converter, resolution: 10 bit) are transmitted in 2 groups of 2 bytes, followed by 2 bytes for adc channel 14 and 2 bytes for the current value of the optional gyroscope module's temperature sensor.

CAN Id 9: The current values of the left and right motor's encoders are transmitted in 2 groups of 2 bytes, meaning number of encoder signals per 100 msec. Additionally, the value of a counter is transmitted, which allows to prove that all encoder messages are received. Because the en-coder values change only every 100 msec but the message is transmitted every 10 msec, there will normally occur 10 messages with the same measurement content, which can be uniquely identified by the counter values.

CAN Id 10: The current state of bumpers 0 - 5 is transmitted in a single byte, containing '0' at the corresponding bit position if the bumper is pressed and '1' otherwise, i.e. 0x3F means "no bumper is pressed". The current state of the remote control's buttons 0 - 7 is transmitted in a single byte, containing '0' at the corresponding bit position if the button is pressed and '1' otherwise, i.e. 0xFF means "no button is pressed".

CAN Id 11: The current position of the robot is transmitted in three values for x-position, y-position and orientation. The units are mm resp. 0.1 * degree. It is assumed that the robot starts at (0 mm, 0 mm, 0.0 degree) when the microcontroller is reset. The position is computed by dead reckoning only.

CAN Id 12: The accumulated values of the left and the right motor's encoders are transmitted in 2 groups of 4 bytes.

CAN Id 13: The current values of the tilt sensor are transmitted in 2 groups of 2 bytes, followed by the current values of the compass sensor's curve 1 and curve 2. The actual direction may be computed by applying a formula of the type: dir = ((pi + atan2(curve1, curve2)) * 180) / pi

CAN Id 14: The current values of the estimation of the robot's angle and its standard deviation are transmitted in 2 groups of 4 bytes. This estimation is based on the data from the optional gyroscope module.

CAN Id 20: The hardware identification represented by an unique number and the firmware version are transmitted in 2 groups of 2 bytes. The current value of the firmware's loop count is transmitted in 4 bytes.

CAN Id 21: The current values of the analog sensors 0 - 3 (attached to channels 0 - 3 of the optional second C167's analog/digital converter, resolution: 10 bit) are transmitted in 4 groups of 2 bytes.

CAN Id 22: The current values of the analog sensors 4 - 7 (attached to channels 4 - 7 of the optional second C167's analog/digital converter, resolution: 10 bit) are transmitted in 4 groups of 2 bytes.

CAN Id 23: The current values of the analog sensors 8 - 11 (attached to channels 8 - 11 of the optional second C167's analog/digital converter, resolution: 10 bit) are transmitted in 4 groups of 2 bytes.

CAN Id 24: The current values of the analog sensors 12 - 15 (attached to channels 12 - 15 of the optional second C167's analog/digital converter, resolution: 10 bit) are transmitted in 4 groups of 2 bytes.

CAN Id.	Data Format [no. of bytes]	Description
0		
1	control_mode[2], settings[6]	control command (depending on control mode)
2	-	reserved (don't use)
3	-	reserved (don't use)
4	hw_id[2], fw_version[2], loop[4]	hardware identification, firmware version, loop count
5	adc_00[2], adc_01[2], adc_02[2], adc_03[2]	analog input channels 0 - 3 (10 bit): sonar sensor 0 - 3
6	adc_04[2], adc_05[2], adc_06[2], adc_07[2]	analog input channels 4 - 7 (10 bit): sonar sensor 4 - 7
7	adc_08[2], adc_09[2], adc_10[2], adc_11[2]	analog input channels 8 - 11 (10 bit): sonar sensor 8 - 9, adc channel 10, adc channel 11
8	adc_12[2], adc_13[2], adc14[2], temp[2]	analog input channels 12 - 13 (10 bit): motor current right and left, adc chan-nel 14, board temperature
9	encoder_left[2], encoder_right[2], enc_count[4]	motor encoder left and right, message count
10	bumpers[1], rc[1]	bumper and remote control states
11	position_x[3], position_y[3], orientation[2]	position as ascertained by odometry
12	enc_odo_left[4], enc_odo_right[4]	accumulated values of left and right motor's encoders
13	tilt_1[2], tilt_2[2], curve_1[2], curve_2[2]	tilt sensor and compass (optional)
14	gyro_angle[4], gyro_sigma[4]	data from optional gyroscope module
15	-	reserved (don't use)
16	-	reserved (don't use)
17	-	reserved (don't use)
18	-	reserved (don't use)
19	-	reserved (don't use)
20	hw_id[2], fw_version[2], loop[4]	hardware identification, firmware version, loop count
21	adc_00[2], adc_01[2], adc_02[2], adc_03[2]	analog input channels 0 - 3 (10 bit)
22	adc_04[2], adc_05[2], adc_06[2], adc_07[2]	analog input channels 4 - 7 (10 bit)
23	adc_08[2], adc_09[2], adc_10[2], adc_11[2]	analog input channels 8 - 11 (10 bit)
24	adc_12[2], adc_13[2], adc_14[2], adc_15[2]	analog input channels 12 - 15 (10 bit)

B Complete source of the singleStep() method

```
bool RcsClient::singleStep(Macs::IBasicControl_var basicControl,
                           Macs::ICamera_var       cameras,
                           Macs::ILaserScanner_var laserScanner){

        // determine resolution:
        Macs::ICamera::CamResolution resolution = Macs::ICamera::VGA;
        Macs::ICamera::RgbImageArray *rgbImage2;

        int w = 640;  // must be consistent with resolution
        int h = 480;  // must be consistent with resolution
        std::vector<Foa> foaList;
        Foa foa_topDown;

        int goalPosX = w / 2;
        int goalPosY = h / 2;
        int currentPosX, currentPosY;
        int diffPosX, diffPosY;
        int changeSteps = 3;
        int signX, signY;
        int posTolerance;
        int maxTrails = 30;
        int currentTrail;
        bool xyPoseReached;

        // configure camera
        char* pStr;
        CORBA::Short res = cameras->configCamera(Macs::ICamera::LEFT,
                resolution, 250, 250, pStr);
        cout << pStr << endl;
        Macs::ICamera::Pan panPos;
        Macs::ICamera::Tilt tiltPos;

        // get image
        Macs::ICamera::RgbImageArray *rgbImage;
        bool success = cameras->getRgbImage(Macs::ICamera::LEFT, rgbImage);
        if (success)
                saveppm(rgbImage->get_buffer(), "test.ppm", w, h);

        // compute focus of attention (Bottom-up)
        Foa f;
        f = vocus((const char*)(rgbImage->get_buffer()), w, h);
        // printFoa(f);
        int n = w * h * 3;
        unsigned char buffer[n];
        for (int i=0; i<n; i++)
```

```
{
        buffer[i] = (*rgbImage)[i];
}
drawFoa(buffer, w, h, f);

// show image
showImage(buffer, "test.ppm", w, h);

// get current focus positions and tolerance value
currentPosX = f.x;
currentPosY = f.y;
if(f.radiusheight > f.radiuswidth)
    posTolerance = f.radiusheight;
else
    posTolerance = f.radiuswidth;

// control loop
currentTrail = 0;
do{
    // compute pan and tilt position of foa: (the parameters are estimated)
    diffPosX = currentPosX - goalPosX;
    diffPosY = -(currentPosY - goalPosY );

    if((abs(diffPosX) < posTolerance)  && (abs(diffPosY) < posTolerance))
    {
        xyPoseReached = true;
    }
    else
    {
        xyPoseReached = false;

        // move camera to foa
        if(diffPosY < 0)
        {
            signY = -1;
        }
        else
        {
            signY = 1;
        }

        if(diffPosX < 0)
        {
            signX = -1;
        }
        else
```

```
            {
                signX = 1;
            }

            cameras->changeCameraPanTilt(Macs::ICamera::LEFT,
                                         (signX*changeSteps),
                                         (signY*changeSteps));

            // get new image:
            success = cameras->getRgbImage(Macs::ICamera::LEFT, rgbImage2);

            // compute new focus of attention (Top-down)
            topdown_search = true;
            topdown_factor = 1;
            foaList.push_back(f);
            foa_topDown = vocus((const char*)(rgbImage2->get_buffer()), w, h, &foaLis

            // new positions
            currentPosX = foa_topDown.x;
            currentPosY = foa_topDown.y;

            // delete memory of the current pic
            delete rgbImage2;
        }
        currentTrail++;
    }while((currentTrail < maxTrails) && (not(xyPoseReached)));

    // get new image:
    success = cameras->getRgbImage(Macs::ICamera::LEFT, rgbImage2);

    // compute focus of attention (Top-down)
    topdown_search = true;
    topdown_factor = 1;

    foaList.push_back(f);
    foa_topDown = vocus((const char*)(rgbImage2->get_buffer()), w, h, &foaList);
    unsigned char buffer2[n];
    for (int i=0; i<n; i++)
    {
        buffer2[i] = (*rgbImage2)[i];
    }
    drawFoa(buffer2, w, h, foa_topDown);

    // show image
    showImage(buffer2, "test2.ppm", w, h);
```

```
        if(xyPoseReached)
        {
            cout << "after "<< currentTrail << " steps object of interest " <<
                "in the center of the picture.\n" << endl;
        }
        else
        {
            cout << "After "<< currentTrail << " steps, the system was not able " <<
                "to focus on the object of interest.\n " << endl;
        }

        delete rgbImage2;
        delete rgbImage;

        return false;
}
```