ROBCL - An Distributed Object-Oriented Robot Programming Language

A. Baumann, B. Baginski and S. Riesner
Chair of Real-Time Systems and Robotics, Department of Computer Science
Technische Universität München
Arcisstr. 21, D-80333 München
email:{baumanna|baginski|riesner}@in.tum.de

Abstract: In this paper we present a client-server based architecture for programming several actors and sensors in a heterogeneous system architecture. To integrate all the different computer systems an object-oriented library builds the communication base. This general communication system (GECOS) provides secure communication via RPC, Shared Memory, CAN or Ethernet. The ROBCL - robot control language is a library providing prototypes for actors and sensors in a robotic laboratory. Using the ROBCL library it is possible to run several programs simultaneously in the same environment, using different or the same devices. Because of a security concept it is not possible that two programs control an actor at the same time by mistake.

Keywords: robot programming, object-oriented robot language, communication layer, control

I. Introduction

At the beginning of robot programming, it was extremely difficult to couple several sensors and actors. Programs for all actors had to be written, which are synchronised over signals. The sensor values had to be evaluated on separate computers and then send to the robotic control. The programming languages, for example V+, contain only very simple data and control structures [Stä94].

Some robot languages, for example RCCL/RCI, permit a substantially more comfortable programming [HL84]. This C-based language makes it possible to link positions or orientations in the world to procedures and to sensor values.

Most robotic laboratorys have a lot of different systems. There are manipulators, cameras, distance measurement sensors, force-torque-sensors, etc. All these sensors have different communication channels and protocols. To control that environment, real-time computers, personal computers or workstations are used. Our goal is to develop a system, which can be programmed using C++ and which is easy to expand by other sensors, actors and computer systems.

Thinking of these requirements it seems to be useful to specify this system in CORBA [OMG98]. However, there is no possibility to itemise the quality of the line between applications within the interface description language IDL [SGHP97]. For closed loop control application it is important to assure that you have a real-time connection between sensor and actor.

II. GECOS

A. Requirements

The goal of the **GE**neral **CO**mmunication **S**ystem is to provide a **media independent communication between different computer systems**. In a robotic laboratory you can classify a communication connection in either a:

- time independent connection or a
- time dependent connection

In the time independent case, e.g. the command to open a gripper, the type of the connection media is less important. It should even be possible to control a robot via satellite. But you need a very fast connection for sensor controlled actions. In this case you have to choose a

real time computer system and communicate via shared memory or CAN Bus.

A further aspect is **the best choice of the communication media**. The system has to select (if you do not specify a communication media) the best possible **quality of service**. If there is no possibility to choose a required communication medium the connection fails.

The **status of a message**, which is sent over the communication medium, must be requestable for the user. One must know for example whether the robot already processed a movement instruction or whether a sensor value request is already answered.

Another important point is the **net security**. No unallowed user must move the robot and it must not be possible to move any actor in the robotic laboratory, if nobody is present in the laboratory.

Since there are actor commands, e.g. move the robot, and sensor commands, e.g. query the position of the robot, a **command security** is necessary. It should be possible that two programs run within the same environment and use the same sensors, but the programs must not attempt to move the same robot at the same time.

B. Realisation

The main modules are the GecosObject object and the GECOS-Request-Broker, which is based on a GecosObject object. The basic GecosObject object components and attributes are shown in fig. 1.

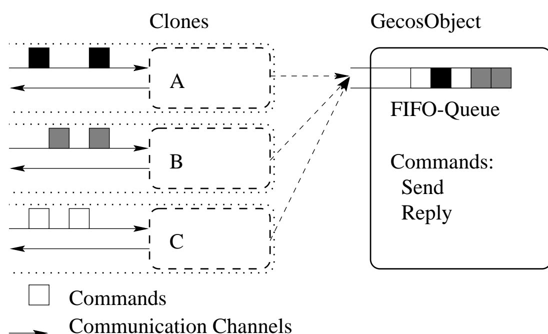

Fig. 1. One module with several communication channels

A GecosObject object can create clones, so it can create communication channels with several partners. All incoming commands of the clones are entered into a First-In-First-Out-Queue. A command object consists of a function code and the function parameters. If the GecosObject knows the function code the appropriated command is executed and the result is returned.

The **GECOS-Request-Broker** handles all the **location independent** communication between several GecosObject objects (see fig. 2). The GECOS-Request-Broker and the GecosObject objects can be executed on different computers. The only requirement is that the GecosObject -objects know where the GECOS-Request-Broker runs.

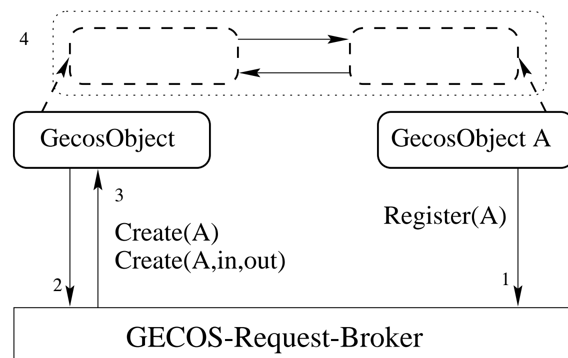

Fig. 2. Communication with an object

In phase (1) the GecosObject object A is registered. Now the GECOS-Request-Broker knows the location of object A. In the **create phase** of an instance of object A the GECOS-Request-Broker is asked (2) about the location of object A. As soon as another object knows (3) the place of module A it can establish a communication (4). If communication channels are added in the creation command `Create(A,in,out)`, the connection is only established if the connection via these channels is possible. To get a secure system, object A can **set host- and user-limits**. Before the communication layer (4) is created, the user and host inclusion of the other object is checked.

The GecosObject object can **support methods** (see fig. 3), which are executed remotely within the connected module. The left GecosObject uses the method

`m_a2` of module A.

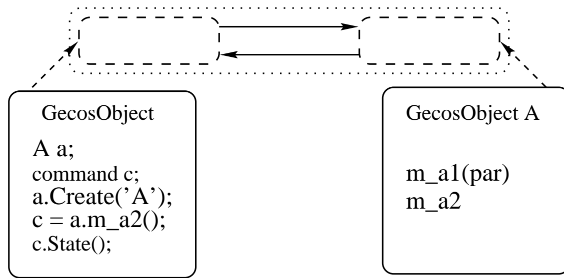

Fig. 3. Using methods of remote modules

After the method or command of an object is called, it is important to know the status of this method. As a command is an object, you can get the status of a command via the method `State`. The status of a command could be returned arbitrarily exactly. A subdivision into **four states** is useful for a robot programming language, in which an instruction can be:

- RUNNING: The command is processed at this moment or is in the queue.
- FINISHED: The command was processed successfully.
- CONNECTION_FAILED: The connection or the server is down.
- COMMAND_FAILED: The command was not executed, due to bad parameters or crash of the server.

Another important part of the method call of distributed objects is the **parameter transfer**. An object (GeneralContents) is developed that forms the base of all communicable data structures. Thus a safe data transfer between processes on different operating systems can be guaranteed, to solve problems such as big and little endian. A parameter of a GecosObject method has to be derived from a GeneralContents, so it has the possibility to encode and decode itself.

There is a large difference between sensor and actor commands. Sensor commands do not change the representation of the devices or move the real device. On the other hand, actor commands move a device or e.g. change the resolution of a frame-grabber (camera) device. It should not be excluded that two programs try to execute actor commands on the same device at the same time. But several programs can test the sensor values of any sensor at the same time without a problem. Neverless one needs a possibility to limit even testing the sensor values for time critical functions like sensor controlled movements. Therefore the **command security** concept differentiates between a *sensor, actor,* and *master status.* In the sensor status sensor values can be read. In the actor status actors can be moved. In the master status all inquiries of other processes are rejected. When a connection to a device is established the process enters the sensor status by default. The actor or master status can be achieved if nobody else is in this status. The actor and master status can be protected with a password, so that all processes knowing this password can enter the same status and move the actors simultaneously. In this case all processes know that there are other processes moving the same device. This feature is very useful, because you could write one program watching the laser distance sensor and taking over the control of the robot if it gets to close to an object. The other program controls the robot if the distance is alright.

C. Example

In this section we want to show that it is very simple to write a server for a laser distance sensor (LDS). The class which represents the LDS looks like this:

```
class LDS : public GecosObject
{ public:
    LDS():GecosObject(){};
     /* constructor */
    command Get (cont_double *d) {;
     /* read the distance of the LDS */
       return Rexec(Get_fc,d);
         /* command init */
    }
};
```

The method `Get` just sends the request for the distance to the laser distance server. This is realised by the `Rexec` function, which generates an object command. The command is sent with the function code `Get_fc`, the parameters and a handler for this command to the server. Because of the function code the server

knows what to do. The handler is necessary, so that the server knows where to send the answer.

The next step is to write a server program which handles the connection to the GECOS-Request-Broker, to the real device and the requests of the clients:

```
#include <LDS.h>
/* distance variable */
static double distance;
LDS prototype;

static void m_Get
 (GecosObject *this, command &c,
  int fc, contents &cs_in) {
 /* implementation of function Get */
  distance = <read value>;
  c->Reply(&distance,FINISHED);
}

int main () {
/* the server */
   /* contact the
      GECOS-Request-Broker */
  prototype.Register('LASER');
   /* provide the method Get
       via the function code Get_fc*/
  prototype.Handle(Get_fc,m_Get,SENSOR);
   /* start the server */
  GecosObject::RUN();
}
```

For a client it is now very easy to read the sensor values:

```
int main () {
   LDS lds;
   cont_double d;
   lds = lds.Create('LASER');
     /* establish connection */
   command c = lds.Get(&d);
     /* get the distance */
   while (c.State()!=FINISHED) {};
     /* wait for value */
   cout << d << endl;
}
```

As you can see in this small example all the methods are evaluated asynchronously. To make sure that you really have the right value in your variable you have to wait for the termination of the command. Here the waiting is realised with polling another way is to suspend the process.

III. ROBCL

A. *Requirements*

According to [SB96] a robot programming language should contain two substantial features. On the one hand **special data types** must be designated, as for example transformation matrices. And on the other hand **device specific functionality for movement** and **for the sensor data acquisition** have to be provided.

To control all devices in a laboratory environment as easy as possible it is useful to have a **common absolute cartesian coordinates system**. All moving instructions refer to this frame of reference. The transfer of a block (see fig. 4) becomes very simple.

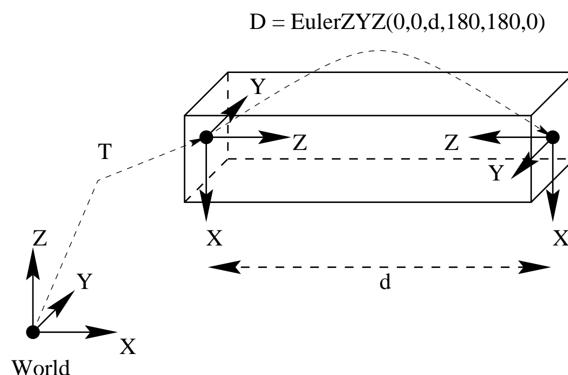

Fig. 4. The transfer positions of a block with two gripper frames. In EulerZYZ representation the first three parameters indicate the shift in x -, y -, and z-direction. The fourth value indicates the rotation around the z-axis, the fifth around the new y-axis and the sixth around the new z-axis.

The start situation is the robot **A** holding the block in its gripper **a**. Now the robot hast to move its gripper to frame **T**. The next step is to move robot **B** with an open gripper **b** to position $T * D$ (matrix multiplication). Now close the gripper **b** and open the gripper **a**. Move the robots back to their start position.

An important aspect in the development of a programming language for robots is the **simple expandability with additional actors and sensors**. It should also be possible to let several devices appear like one. If one thinks of sensor controlled accessing, then the robot, the force-torque-sensor, and the gripper should conjoin to one virtual device. As we have shown in the last section it is very easy to write new servers with the GECOS library. In this section we focus on the

analysis of typical devices in a robotic laboratory and what kind of servers have to be written.

B. Realisation

The **ROB**ot **C**lass **L**ibrary is an object-oriented language, based on GecosObject and the frames of [FTB+94]. For a client-server architecture in a robotic laboratory it is useful to represent each device by one server. As we remarked, it is possible to write one server for several devices to provide methods like grip the block at position A. But to write a server with several devices is only useful if you have hard real time requirements. Otherwise you would provide this in a function or method.

The basic class for all devices is the Device class which adds **the position and orientation of a device in the world** to the GecosObject object.

In fig. 5 **typical devices of our robot laboratory** are represented. Altogether there are two industrial robots, two force torque sensors (FTS), four laser distance sensors (LDS), two grippers and a frame grabber device.

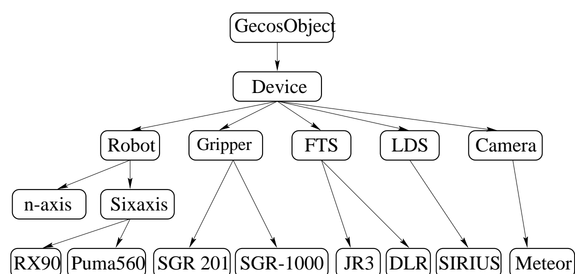

Fig. 5. Devices in a robot laboratory

The most interesting devices in a robotic laboratory are the robots. In our case there are only manipulators. The manipulators have a fixed position in the world. The abstract manipulator has an unknown number of joints. The *sensor* values of the robots are the position and the orientation of the tool frame in the world, and the configuration of the robot (e.g. the configuration of the robot RX90 is a triple out of (lefty|righty, above|below, flip|noflip)). The *actor* commands can set the configuration and move the robot. The move command consists of two parts. The first parameter is the goal position of the tool frame. The second parameter defines the kind of the trajectory between the current and the goal position. The basic trajectory types are *straight* and *joint* movement. You can expand both types by the value *break* or *nobreak*. The *break* parameter moves the tool frame exactly to the goal frame, unlike the *nobreak* command which interpolates between the goal frame and the following frame (if there is one). For sensor controlled movement an alter-move is provided. The parameter of this command is a relative frame which is multiplied to the current position of the robot within one system-clock interval of the control unit.

On next specification level there are the n-axis manipulators. For our laboratory we only need the six-axis manipulator class. Additional *sensor* and *actor* commands permit the request of the joint values and the movement specified by joint values.

The final level presents the exact robot, e.g. RX90, Puma560, or simulation. For these real devices server have to be implemented, so all methods of the parent classes can be used.

For a user who wants to move the robot only with frame movements this hierarchic structure is very useful, because you can use the same program for all robots. You do not have to make the decision which real robot has to do the moves until the `Create('robot')` command. Now you could ask for the RX90, but if the RX90 robot is not online and the connection fails you could ask for another robot within the same program. The other devices are quite simple so we do not specify the server structures more exactly for these sensors.

The next important parts of a robot programming language are the **data structures**. In fig. 6 the most important data structures which occur in a robot laboratory are represented.

A central data structure is the homogeneous matrix which can be used to store the position and orientation of a frame in the world. Moreover you can per-

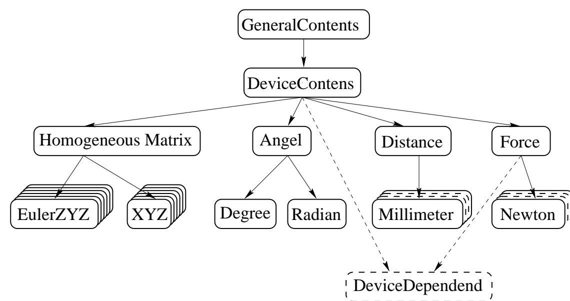

Fig. 6. Data structures for actors and sensors

form the basic math operations like multiplication and inverse on these matrices. Furthermore the matrices can be converted between the different representations (CEulZYZ, CXYZ, ...). Now it is very easy to calculate a position in the world and then move the robot to this position.

C. Example

In this example we want to show how easy it is to program the devices in our robot laboratory. The scenario is a robot gripping a block. The block's grip point is at the position `BLOCK` in the world.

```
#include <sixaxis.h>
#include <gripper.h>

int main () {
   CRobot myR;
   CGripper myG;
   command c;

   /* establish the connection */
   myG.Create('ROBOT');
   myR.Create('GRIPPER');
   /* set the robot position*/
   myR.Mount(ROBOT_BASE);
   /* get actor status */
   myG.actor();
   myR.actor();
   /* move the robot to a point
      20 cm over the block */
   c=myR.Move(TransMat(0,0,-200)*BLOCK);
   /* open the gripper to 10 cm */
   while (myG.Open(100).State()
         ==RUNNING) {};
   /* wait for the robot */
   while (c.State()==RUNNING) {};
   /* move down */
   while (myR.Move(BLOCK)==RUNNING) {};
   /* close gripper with force 30% */
   while (myG.Close(30).State()
         ==RUNNING) {};
   /* move up */
   myR.Move(TransMat(0,0,-200)*BLOCK);
}
```

There is no trouble shooting done in this short program. Normally you would have to check whether the connection was established and whether you got the actor commands right.

IV. Conclusions

We presented a powerful robot control language which makes it very easy to write programs with several devices. Because of the communication library GECOS it does not matter on which operating systems the server or client processes run. We showed that it is very simple to expand our system by more servers and devices.

The practical experiences with our implementation are very promising. The system is used in the practical robotics laboratory course successfully, as well as in further projects where real time trajectory generation and control is integrated.

Acknowledgement

The first and second author would like to particularly thank the third author, who carried out the entire base implementation. Furthermore we would like to thank our students, who used the system and put it through its paces.

References

[FTB+94] Raphael A. Finkel, Russell H. Taylor, Robert C. Bolles, Richard P. Paul, and Jerome A. Feldman. AL, a programming system for automation. Technical Report CS-TR-74-456, Stanford University, Department of Computer Science, 1994.

[HL84] V. Hayward and J. Lloyd. *RCCL User's Guide.* McGill University, Montreál, Québec, Canada, 1984.

[OMG98] Object Management Group, Inc., 492 Old Connecticut Path, Framingham, MA 01701. *The Common Object Request Broker: Architecture and Specification*, 2.2 edition, feb 1998.

[SB96] H.-J. Siegert and S. Bocionek. *Robotik: Programmierung intelligenter Roboter.* Springer-Verlag Berlin Heidelberg New York, 1996.

[SGHP97] D. Schmidt, A. Gokhale, T. Harrison, and G. Parukar. A high performance end system architecture for real-time CORBA. *IEEE Communications Magazine*, 35(2):72–78, 1997.

[Stä94] Stäubli. *V+ Language Version.* Adept Technology, Faverges, France, version 11.0 edition, 1994.