Reactive Real–Time Programming with Distributed Agents

Gerhard Schrott

Technische Universität München
Institut für Informatik
D-80290 Munich, Germany

Abstract. The proposed reactive real–time programming system is a new approach to implement complex distributed heterogeneous real-time applications. It is based on the notion of distributed multi–agent systems. The whole control task is decomposed top down into small execution units, called agents which communicate by sending and executing contracts and are specified in a hardware independent language based on states and guarded commands. At compile time the agents are distributed to specified targets. PC's, micro-controllers, programmable logic controllers and even programmable logic devices are supported. The system automatically translates each agent to the particular code and realizes the communication including a bidding protocol between the agents either on the same processor or within a network. Due to a strictly cyclic processing of the agents exact response times can be guaranteed. Zero delay agents can be implemented in hardware.

1 Introduction

Reactive programming of distributed systems connected by a field–bus is nowadays imperative in process control. Networks comprising PC's and/or micro-controllers are available at low prices and will replace the "one for all" computer solution based on conventional real-time operating systems and will solve its inherent problems: Micro-controllers dedicated to time–critical applications guarantee the needed reactiveness; the complexity of control applications is mastered by a natural distribution of tasks; modern programming paradigms are used to improve the flexibility and fault tolerance of the system.

1.1 Multi–agent systems

Agents [4] are small autonomous software units which are able to perceive, to plan, to communicate with each other, to decide and to act. Multi–agent systems (e.g. [1],[5]) are in many respects similar to distributed real–time systems. Tasks correspond to agents, messages to contracts between agents, perception and action are equivalent to the input/output of the technical process and planning and deciding is implicitly programmed in intelligent autonomous applications. It is obvious that the paradigm of an agent can easily be transfered to distributed real–time systems and may give new impulses to the programming of distributed process control.

1.2 Cyclic predictable scheduling

Cyclic scheduling was already used in the earliest real–time systems. It is still applied to safety critical applications. Lawson [3] stresses that only cyclic systems can be proved in their time behavior.

1.3 Synchronous programming

Synchronous languages as ESTEREL, LUSTRE or SIGNAL ([2]) provide statements which allow a program to be considered as instantaneously reacting to external events. Their behavior is also fully deterministic, however their computational power is only equivalent to finite automaton. They create reactive kernels which need additional layers for physical input/output, for data management and for communication between different kernels.

1.4 Real–time programming with distributed agents

Most available operating systems are based on the execution of parallel tasks causing the known problems of scheduling and interrupt response times. In this paper a new approach for programming distributed heterogeneous real-time applications is proposed based on the notion of distributed multi-agent systems. The communication between these agents is transparent to the programmer. The system guarantees response times by a strictly cyclic and therefore predictable scheduling. Using a restricted subset of the system specification even the hard real-time requirements of synchronous programming are met: the resulting code is the definition of a finite automaton specified in VHDL.

2 MAD–RTS Specification

The proposed reactive multi–agent distributed real-time system (MAD–RTS, [6]) combines specification and coding of the control programs in a top down approach. It supports the definition of small agents which communicate with each other by sending contracts to start activities of other agents. If more than one agent can perform the needed activity, bids are sent and the 'cheapest' agent will get the contract. Each agent has a set of defined states and executes the guarded actions of this state. It interacts via generic sub-agents with physical input/output channels, timers and other specific hardware. As in object oriented programming the agent executes the contract like a method without showing the real implementation.

A complete MAD program starts with the declaration of the available target processors and is followed by a number of agents (see examples in chapter 4). The complete definition of an agent in MAD is divided into the following parts:

- Declaration of the target microprocessor, the agent will be executed on
- Instantiation of sub-agents used by the agent to interface to the hardware and to special functions

– Declaration of bids for contracts
– Declaration of contracts, the agent will accept
– Action part

2.1 Target definition

The target processor chosen within the network to execute the agent is defined after the keyword `target`. More than one agent may of course run on one processor. The final distribution is determined both by the physical input/output channels used by the agent and connected to the processor, and by the load on one processor resp. the response time needed by the agent. This assignment can be changed at any time; only recompiling is necessary to get a new running system.

2.2 Sub-agents

At lowest level generic sub-agents are defined to realize the interface to the technical process (e.g. digital and analog input/output, timer, stepper motor, pid-controller). They are instantiated in the `decls` definition of a MAD–program with the actual i/o-address and mnemonic identifiers to enhance self documentation of the program. Consequently, local variables and arrays are also declared as sub-agents. Sub-agents written in C may be added to the libraries.

2.3 Contracts and bidding

The only interface between agents is the contract protocol. In the `contracts` definition every contract which can be called by other agents is listed. A contract transfers parameters and causes the execution of actions or in most cases a state transition in the agents action part. Included in the communication system is a contract net bidding protocol. If more than one agent can execute a contract each agent sends its cost to perform the activity of the contract, computed by a cost function supported in the `bids` definition. The runtime system chooses the 'cheapest' agent and sends the contract to it.
Sending a contract only starts the activity of another agent but does not wait for its completion. Therefore, deadlocks in contract calls cannot occur. It is the duty of the programmer to avoid cyclic dependencies of calls to contracts e.g. by using strictly hierarchical dependencies of contracts.

2.4 States and actions

The action part is subdivided into a set of states. Control tasks usually change between different states of operation, e.g. at lowest level 'on' or 'off', at higher level 'open', 'closed' or 'error'. One state is active and the actions comprising it are executed. Each action is bound by a condition (guard) and only if it is true the corresponding statements are executed. At lower level these conditions will be

signals from the controlled process, at higher level it may be timers or conditional expressions on variables. There is also a special condition `once` to ensure that the following statements are executed only once when entering the corresponding state. All other conditions in one state are tested cyclically and all agents fixed to one processor are executed one after the other to guarantee an exactly predictable time behavior. Special agents may be defined for time consuming computations which get a specified time slice per cycle. Two distinguished states are obligatory on each agent: At start up the agent goes into the `initial` state, in case of emergency the `shutdown` state is entered.

3 Implementation

The programming system MAD–RTS is hosted on a PC with MS-DOS. It contains the compiler for MAD and code generators and run time systems for different targets. The compiler is written in the object-oriented language C++. The syntax of MAD–RTS is defined in YACC, for lexical analysis the tool 'FLEX', for syntactical analysis the tool 'BISON' is used. Therefore, the compiler can easily be extended by additional language features as well as additional code generators and sub-agents. The compiler uses the contract specification to generate automatically the code for the transmission of contracts between agents on the same or on different targets. A small run time kernel is added which executes sequentially all agents on one target and realizes the physical communication in the network.

Code generators are available for Intel80x86, MC68HC11 and programmable logic controllers; the communication link is implemented for RS 232 serial link and the CAN–field–bus. If only digital input/output, no numeric expressions and no parameter passing to contracts are used, a further code generator is available which translates an agent program into a table of a finite automaton [7] which can easily be transcribed into VHDL and put into the hardware of a programmable logic device (PLD).

Figure 1 shows the generation of a MAD–RTS application, a small plant with one robot, a machine tool and a conveyer belt. At source level the target of each agent is defined. The agents are compiled into intermediate code which is translated according to the target definition to the dedicated hardware and linked with the run time kernel and the needed libraries. The resulting code is then loaded into the target or burned into a PLD. The resulting distributed multi-agent system is now ready to run; the agents communicate via CAN-Bus or RS232.

No explicit programming of this communication between agents is necessary. Each agent can be shifted to other hardware with only minor changes in the definition part of the agent. Likewise, the communication links are automatically altered without the need for changes in the application program. A contract net protocol with simple bidding is embedded into the runtime system kernel. Specification and coding of the control programs are closely related. The system engineer can design the control application at first according to problem ori-

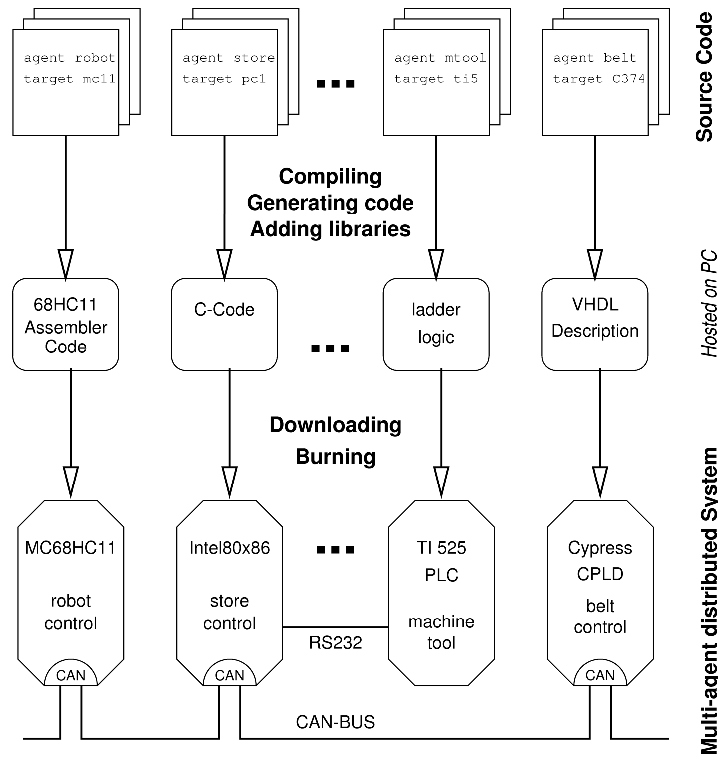

Fig. 1. Generation of a MAD–RTS application

ented criteria and afterwards decide about the optimal hardware structure for his application.

4 Programming examples

To test the MAD–System on a real platform we use a FischerTechnik model of a twin elevator with four floors. The model has the complete functionality of a twin elevator with all keys, switches, lamps, displays and motors. It is controlled by one microprocessors MC68HC11 on each floor, one in each cage and one at each motor platform, all connected via the CAN–field–bus (see figure 2).

4.1 Door agent

The implementation of the door agent shows an example of a MAD program. Every agent may send the contract `open` in order to induce the door agent to open the elevator door, wait for 10 seconds and close it again. If the light barrier is interrupted during closing, the door is opened again. If the door is closed, the contract `start` is sent to agent elevator1.

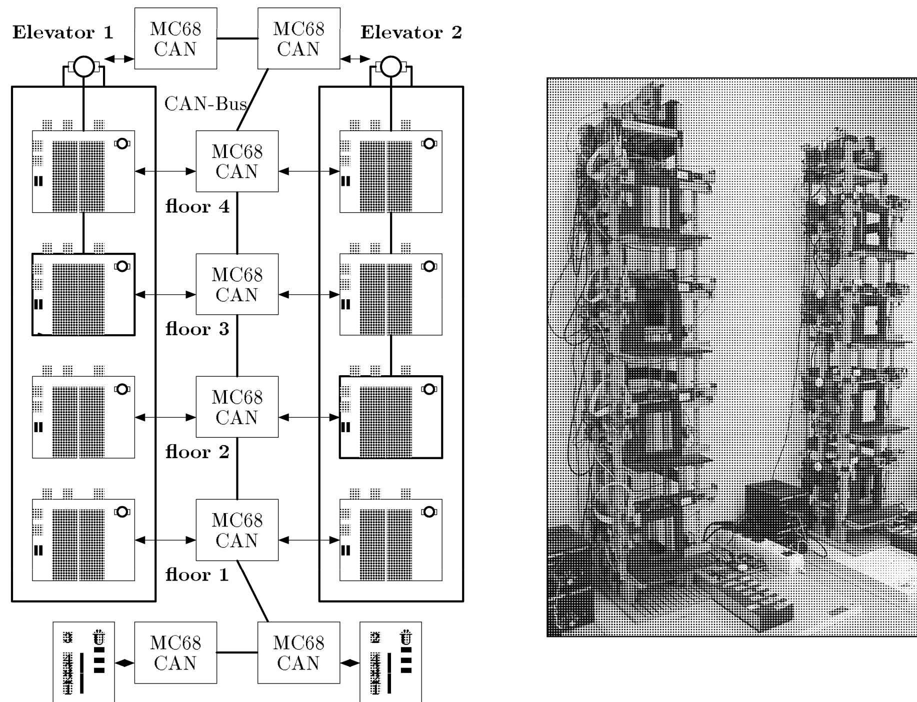

Fig. 2. Microprocessor network for twin elevator and a View of the model

```
targetdecl mc68hc11 mcfloor2; % target definition
agent door2 target mcfloor2;  % door agent elevator1, 2nd floor
decls                         % declaration of subagents
  DigOut motor(out1,on,off);
  DigOut direction(out2,open,close);
  DigIn  open_key(in1,on,off);
  DigIn  closed_key(in2,on,off);
  DigIn  light_barrier(in7,interrupted,ok);
  Timer  delay;
contracts                     % accepted contract
  open do newstate opening;
states                        % start of action part
  closed/shutdown:
    once          => motor.off;
  opening:
    once          => {direction.open; motor.on;}
    open_key.on   =>  newstate waiting;
  waiting:
    once          => {motor.off; delay(10000);}
    delay.tout    =>  newstate closing;
  closing/initial:
```

```
    once           => {direction.close; motor.on;}
    closed_key.on  => {newstate closed; elevator1.start;}
    light_barrier.interrupted
                   => {motor.off; newstate opening;}
endagent;
```

4.2 Bidding agent

There exist two similar agents each controlling one elevator motor. A contract may be sent to both elevator agents if a cage is called from a certain floor. Both elevator agents compute their actual cost to go to this floor based on their actual state and the difference to the floor the cage is actually located. Of course, more sophisticated cost functions are necessary to get a better strategy. The runtime system will send the contract to the cheaper agent and it will send the cage to this floor. The following example shows the relevant parts for bidding in the definition of the contract:

```
targetdecl mc68hc11 mcelev1;     % target definition
agent elevator1 target mcelev1;  % motor agent elevator1
decls                            % declaration of subagents
  varinteger aktfloor;           % for a local variable
  arraybool(4) stop;             % and a local array
bids                             % cost function for contract get
  get (integer floor)
     cost ((instate busy)*4 + abs(floor-aktfloor));
contracts                        % accepted contract
  get (integer floor)
     do stop(floor).set(true);
states                           % start of action part
  .
    busy:                        % motor busy
  .
endagent;
```

An agent controlling the keys at the second floor will send the following contract to call one of both elevators:

```
elevator1|elevator2.get(2)
```

5 Results

Due to the strictly cyclic processing you get the advantages of a distributed multi–agent programming system combined with the exact preview of the maximal delay time until the acceptance of a new signal. For hard real-time requirements a 'hardware agent' with zero delay can be created using the VHDL code generator. The strict cyclic approach produces a minimum of overhead and

results in fast reaction times; on the micro-controller MC68HC11 (8 MHz) the periodic execution time for the model elevator is between 3 to 10 msec, including the transfer on the CAN-Bus. The produced code is very small, max. 20 KBytes for one controller.

The complex program can be easily tested: A monitor shows the actual states of each agent and the contracts sent between the agents. Reaction on errors can be embedded into the guarded actions and fault tolerance can be realized by distributing tasks on different hardware. Our experience showed that even severe errors in one agent don't lead to a total breakdown of the controlled process.

6 Conclusions

Reactive real–time programming with distributed agents covers a wide range of distributed real-time applications, supporting conventional PC based applications, micro-controllers and programmable logic controllers, down to hardware/software codesign. It introduces the notion of multi–agent systems, the security of state driven design and the flexibility of distributed multi–processing. Because of its strictly cyclic execution, exact response times can be guaranteed. Problems with priorities, task scheduling and interrupts don't occur. A redesign of the system is planned embedding the MAD-RTS into the object oriented language JAVA. This promises more flexibility, portability and a wider acceptance.

References

1. S. Hahndel and P. Levi: A Distributed Task Planning Method for Autonomous Agents in a FMS. *Proc. IEEE/RSJ/GI Int. Conf. on Intelligent Robots and Systems (IROS '94) Sept. 12-16, 1994, München, Germany.* Los Alamitos, CA: IEEE Computer Society Press, pp. 1285–1292, 1994..
2. Halbwachs, N.: Synchronous Programming of Reactive Systems. Kluwer Academic Publishers, Dordrecht 1993.
3. H.W. Lawson: Parallel Processing in Industrial Real-Time Applications. Prentice Hall, Englewood Cliffs, NJ., 514 p., 1992.
4. J. P. Müller: The design of intelligent agents. Lecture notes in computer science; 1177 : Lecture notes in artificial intelligence, Springer, Berlin, 227 p., 1996.
5. G. Schrott: An Experimental Environment for Task-Level Programming of Robots. *Proceedings of the 2nd Int. Symposium on Experimental Robotics*, Toulouse, Juni 25-27, 1991. R. Chatila (Ed.), Lect. Notes in Control and Information Sciences 190, Springer, Berlin, pp. 196-206, 1992.
6. G. Schrott: A Multi-Agent Distributed Real-Time System for a Microprocessor Field-Bus Network. *Proc. of 7th Euromicro Workshop on Real-Time Systems, Juni 14–16, 1995, Odense, Denmark.* IEEE Computer Society Press, Los Alamitos, California, pp. 302–307, 1995.
7. G. Schrott and T. Tempelmeier: Putting Hardware-Software Codesign into Practice. *Proc. of 22nd IFAC/IFIP Workshop on Real-Time Programming, Sept 15-17, 1997, Lyon, France,* to be published.

This article was processed using the LaTeX macro package with LLNCS style