A Multi-Agent Distributed Real-Time System for a Microprocessor Field-Bus Network

G. Schrott

Technische Universität München
Institut für Informatik
80290 Munich, Germany

Abstract
A new conception of a Multi-Agent Distributed Real-Time System (MAD–RTS) is presented comprising a compiler, operating system kernel and communication tools. It allows for an uniform programming of complex process control systems on a microprocessor field–bus network. The key idea is to decompose the whole control task into small execution units, called agents which communicate by sending and executing contracts. The specification of the agents is done in an hardware independent language using the notion of states and guarded commands. At compile time the agents are distributed to specified target microprocessors. The system automatically translates each agent to the particular code and realizes the communication between the agents either on the same processor or through the field–bus.

1 Introduction
Distributed systems on a microprocessor network connected by a field–bus are increasingly used in process control. Such systems now available at low prices will replace the "one for all" computer solution and solve its inherent problems: Dedicated computers guarantee the needed response times in time–critical applications; the complexity of control applications is mastered; modern concepts to improve the flexibility and fault tolerance of subsystems may be used.

Due to bus standards, heterogenous hardware can be easily connected. However, adequate tools for specification and programming of distributed microprocessor systems are missing. Even if the connection of heterogenous hardware is supported by commercially available real-time operating systems the links have to be programmed explicitly using ports or messages and subroutines to access the field–bus. All available operating systems are based on the paradigm of parallel tasks causing the known problems of scheduling and interrupt response times. In this paper a new approach for programming real-time applications is proposed which guarantees response times by a strictly cyclic and therefore predictable scheduling. It does not aim to applications with hard real-time constraints in the range of microseconds, however it can be used by a lot of applications in industrial plants and computer integrated manufacturing, but also by smaller

applications (e.g. machine tools, elevators, cluster of embedded systems).

1.1 Multi-agent systems
One of the main topics of distributed artificial intelligence is the conception of distributed multi–agent systems ([1],[3],[6]). Agents are small autonomous units which are able to perceive, to plan, to communicate with each other, to decide and to act. Multi-agent systems are in many aspects 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 the agent can easily be transferred to distributed real–time systems and may give new impulses to the programming of distributed process control.

1.2 Cyclic predictable scheduling
Cyclic scheduling has already be proposed in different papers on real–time systems: Kopetz [4] compares the properties of event-triggered and time-triggered distributed real-time systems, underlining the advantages of time-triggered systems; Faulk and Parnas [2] give an algorithm to transform a conventionally programmed application into a cyclic predictable program. Lawson [5] stresses that only cyclic systems can be proved in their time behavior and are therefore admitted for safety critical applications. The proposed multi–agent distributed real-time system (MAD–RTS) is a new approach to program complex distributed heterogenous real-time applications [7]. It also highly supports structured design and test- and maintainability of the resulting system.

2 MAD–RTS Specification
In MAD–RTS specification and coding of the control programs are closely related. The MAD-language 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 subagents
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 microprocessors and is followed by a number of agents (cmp. appendix A). 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 subagents used in 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 host. 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 Subagents

At lowest level generic subagents 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. Subagents are called similar to contracts in other agents. Additional subagents may be added on demand, e.g. C-function call or interface to files.

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 choses 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.

![Diagram of Microprocessor network for twin elevator](Figure 1: Microprocessor network for twin elevator)

2.4 States and Action

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 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 microprocessor are executed one after the other to guarantee an exactly predictable time behaviour. 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 under 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 (see appendix A) 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 subagents. 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. Code generators are available for AT486, MC68HC11 and ladder logic for PLC; the communication link is implemented for the CAN–field–bus.

4 Applications

Two process models built with FischerTechnik bricks are used to test the MAD–RTS system:

1. A twin elevator with four floors (see figures 1–3) 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.

2. A big plant (see figure 4 and figure 5) comprising of an input and an output conveyor belt, four simulated machine tools, two robots for transportation and two stores. Till now it was controlled by a VAX–ELN and a PLC T500; the new distributed control structure includes 5 microcontroller boards, 3 programmable logic controllers, and an AT486 connected via CAN–field–bus.

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.

```plaintext
hostdecl mc68hc11 mc12;
agent door2 host mc12;

decls
DigOut motor(A1,on,off);
DigOut direction(A3,open,close);
DigIn open_key(C1,on,off);
DigIn closed_key(C2,on,off);
DigIn light_barrier(C7,interrupted,ok);
Timer delay;

contracts
  open do newstate opening;

states
  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;}

sndagent;

4.2 Bidding agent
There exist two similar agents each controlling one
elevator motor. A contract may be sent to both el-
levator 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 dif-
ference to the floor the cage is actually located. Of
course, more sophisticated cost functions are nec-
dessary 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:

hostdecl mc68hc11 mc1;
agent elevator1 host mc1;
decis
  varinteger aktfloor;
  arraybool(4) stop;

bids
  get (integer floor) cost
     (((newstate busy)*100 +
       abs(floor-aktfloor)*20);

contracts
  get (integer floor) do
    stop(floors.set(true);

states
   .
   .
   .
   busy:
   .
   .

sndagent;

Some other agent will send the following contract
to call one of both elevators to the second floor:
elevator1|elevator2.get(2)

5 Results
Due to this strictly cyclic processing you get the
advantages of a distributed multi-agent programming
system combined with the exact preview of the max-
imal delay time until the acceptance of a new signal.
The complex program can be easily tested: A moni-
tor shows the actual states of each agent and the con-
tracts sent between the agents. Reaction on errors can
be embedded into the guarded actions and fault tol-
erance can be realized by distributed tasks on different
hardware. Our experience showed that even severe er-
ers in one agent don’t lead to a total breakdown of
the controlled process.

The strict cyclic approach produces a minimum of
overhead and results in fast reaction times; on the
microcontroller MC68HC11 (8 MHz) the periodic exe-
cution 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.

6 Conclusions
MAD–RTS presents a new approach to program
distributed real-time applications. 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, ex-
act response times can be guaranteed. Problems with
priorities, task scheduling and interrupts don’t occur.

No explicit programming of the communication be-
tween the agents is necessary. MAD–RTS is avail-
able on MC68HC11 microcontroller, A1486 and pro-
grammable logic controllers. Each agent can be shifted
to other hardware with only minor changes in the def-
inition part of the agent. Likewise, the communica-
tion 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 of MAD–RTS. The system engineer
can design the control application at first according
to problem oriented criterias and afterwards decide
about the optimal hardware structure for his applica-
tion.

Figure 4: View of the modell plant
Acknowledgements
I want to thank all students who have contributed to design and implementation of the MAD-RTS system, especially B. Sterzbach, G. Rötzter, Ch. Hüsch, A. Schmieja and T. Lopatic.

References


A  Formal grammar of MAD-language

Excerpts of MAD-language specification in Backus-Naur-syntax:

\[
\begin{align*}
MAD_{program} & \rightarrow MAD_{program} \text{ block} \\
& \quad \mid \text{ block} \\
\text{block} & \rightarrow \text{ HOSTDECL hosttype identifier} \\
& \quad \mid \text{ AGENT identifier HOST identifier} \text{ agent_block ENDAGENT} \\
\text{hosttype} & \rightarrow \text{ MC68HC11} \\
& \quad \mid \text{ AT486} \\
& \quad \mid \text{ PLC} \\
\text{agent_block} & \rightarrow \text{ agent_block agent_def} \\
\text{agent_def} & \rightarrow \text{ DECLS declarations} \\
& \quad \mid \text{ BIDS bids} \\
& \quad \mid \text{ CONTRACTS contracts} \\
& \quad \mid \text{ STATES states} \\
\text{bids} & \rightarrow \text{ bids bid} \\
& \quad \mid \epsilon \\
\text{bid} & \rightarrow \text{ identifier form_params COST expression} \\
\text{contracts} & \rightarrow \text{ contracts contract} \\
& \quad \mid \epsilon \\
\text{contract} & \rightarrow \text{ identifier form_params DO action} \\
& \quad \mid \text{ type identifier form_params RETURN expression} \\
\text{form_params} & \rightarrow \{ \text{ type_id_list } \} \\
\text{type_id_list} & \rightarrow \text{ type identifier} \\
& \quad \mid \text{ type_id_list , type identifier} \\
\text{declarations} & \rightarrow \text{ declaration declarations} \\
\text{declaration} & \rightarrow \text{ generic_subagent identifier opt_id_list} \\
\text{opt_id_list} & \rightarrow \{ \text{ id_list } \} \\
& \quad \mid \epsilon \\
\text{id_list} & \rightarrow \text{ identifier} \\
& \quad \mid \text{ id_list , identifier} \\
\text{states} & \rightarrow \text{ states state} \\
& \quad \mid \epsilon \\
\text{state} & \rightarrow \text{ state_id identifier : actions} \\
\text{state_id} & \rightarrow \text{ state_id identifier /} \\
& \quad \mid \epsilon \\
\text{actions} & \rightarrow \text{ actions action} \\
& \quad \mid \epsilon \\
\text{action} & \rightarrow \text{ bid_list identifier , identifier opt_expr_list} \\
& \quad \mid \text{ NEWSTATE identifier ;} \\
& \quad \mid \text{ \{} \text{ actions } \} \text{ expression => action} \\
& \quad \mid \text{ ONCE => action} \\
\text{bid_list} & \rightarrow \text{ bid_list identifier} \\
& \quad \mid \epsilon \\
\text{opt_expr_list} & \rightarrow \{ \text{ expr_list } \} \\
& \quad \mid \epsilon \\
\text{expr_list} & \rightarrow \text{ expr_list expression} \\
& \quad \mid \epsilon \\
\text{expression} & \rightarrow \text{ expression operator expression} \\
& \quad \mid \text{ ( expression )} \\
& \quad \mid \text{ INSTATE identifier} \\
& \quad \mid \text{ unsigned_expression} \\
\end{align*}
\]